AzeoTech® DAQFactory®

DAQFactory - LabJack Application Guide

U3 / U6 / UE9x

9/18/2011

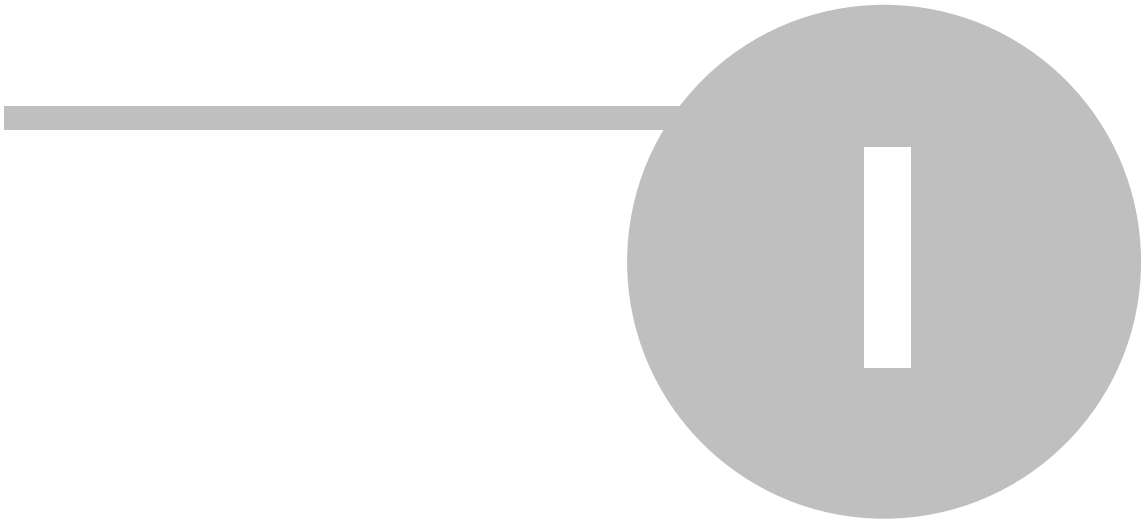
DAQFactory - LabJack Application Guide

DAQFactory for Windows, Version 5.86, Sunday, September 18, 2011:

Information in this document is subject to change without notice.

Table of Contents

1 Introduction



1 Introduction

1.1 Welcome

Congratulations on the purchase of your new LabJack. Included on your installation CD is a trial of DAQFactory-Pro that will convert to a fully licensed copy of DAQFactory-Express after 25-days which will help you make the most of your new device.

With DAQFactory-Express you can take data and control outputs, log data to files easily read by other programs like Excel, create your own screens for displaying your data using any combination of 10 screen components including buttons, graphs, images and more.

For more advanced applications, consider one of the other versions of DAQFactory which, depending on the version, include 42 screen components, networking, PID, alarming, a 3800 image library, unlimited channels, unlimited pages, unlimited scripting and much more. All your DAQFactory-Express applications will work in the other versions of DAQFactory.

This document will start you on your way using your new device with DAQFactory-Express and will also work in all the other versions of DAQFactory. Feel free to use DAQFactory-Express to its full capabilities. If you need a more powerful version such as demonstrated by the trial version, please visit www.azeotech.com or your LabJack reseller to purchase a license.

This document is not for the LabJack U12, which uses a different driver. Please see the DAQFactory User's Guide for information on using the U12 with DAQFactory.

1.2 How to use this guide

This document explains how to do the most common things with DAQFactory and your LabJack device. Because it is likely that you will actually want to do the things described in the first eight chapters, we strongly recommend you actually go through the first eight chapters. After that, you will probably want to jump to the appropriate section in the later part of the manual that describes how to perform more specific tasks with your LabJack device.

Most of the sections include a sample document that you can load directly into DAQFactory. These applications are included in the LJGuideSamples directory of your DAQFactory installation. The exact file name is specified on each page.

1.3 DAQFactory Versions

	PRO	STANDARD	BASE	LITE	STARTER	EXPRESS
Acquisition and logging	Yes	Yes	Yes	Yes	Yes	Yes
Custom Screens	Yes	Yes	Yes	Yes	3 max	2 max
Graphing / Trending	Yes	Yes	Yes	Yes	Yes	Yes
Data Analysis	Yes	Yes	Yes	Yes	Yes	Yes
40 Screen Components	Yes	Yes	Yes	Yes	Yes	Only 11
Sequences and Automation	Yes	Yes	Yes	Yes	Yes	Yes
Email Out / FTP Upload	Yes	Yes	Yes	Yes	Yes	
PID Loops	Yes	Yes	Yes			
ODBC Database Logging	Yes	Yes				
3,800 Image Library	Yes	Yes				
Networking and Web Server	Yes	Yes				

	PRO	STANDARD	BASE	LITE	STARTER	EXPRESS
Advanced Data Analysis	Yes	Yes				
Alarming	Yes					
Auto-Dialer	Yes					
Maximum Channels / Tags	Unlimited	Unlimited	64	32	16	8

2 Installing and Starting

2 Installing and Starting

2.1 Installation

The first step to using your LabJack device is to insert the CD that came with your LabJack and install the included software. The LabJack software must be installed on any machine that uses the LabJack device, and should be installed before you plug the device into the USB port. DAQFactory is a separate installers that should be automatically run once the LabJack drivers are installed. DAQFactory installers do not include the low level LabJack drivers, so you must install the drivers that come from LabJack to use their devices. This is required even if you are going to connect over Ethernet to an Ethernet compatible LabJack device, such as the UE9, unless you plan on using Modbus. If you are going to use Modbus, please refer to the Modbus Guided Tour and the DAQFactory User's Guide.

If you have lost you CD, you can always download DAQFactory from the AzeoTech website at www.azeotech.com. Just download the trial. Likewise, the LabJack drivers should be available from the LabJack website at www.labjack.com.

Note: If you get a alert message in DAQFactory saying that it is unable to load the LabJackN.dll file then you probably forgot to install the LabJack drivers from LabJack. Reinstall from your CD, or go to www.labjack.com for the latest drivers. If you upgrade DAQFactory (by going to www.azeotech.com) or DAQFactory Express (by going to www.daqexpress.com), you will most likely also need to ensure that you are using the latest LabJack drivers. You should therefore go to www.labjack.com and download and install those drivers as well.

2.2 Setup device

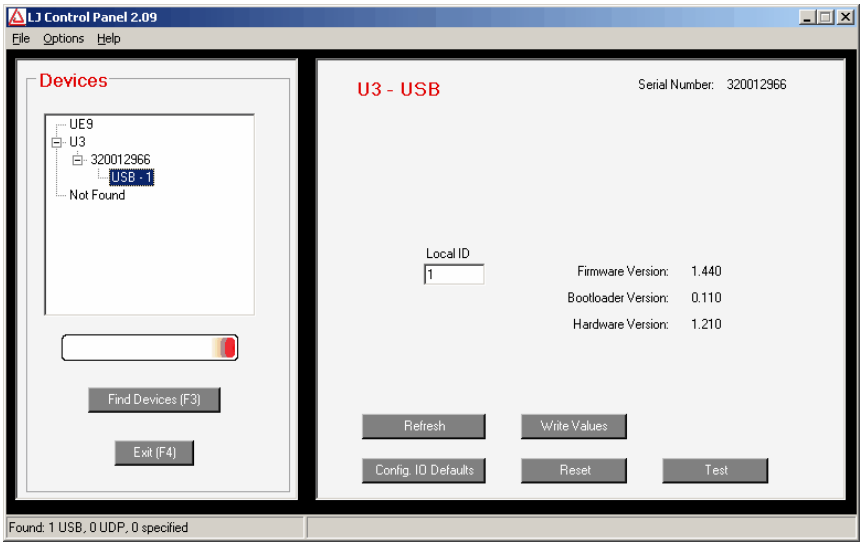
Once installed, and if necessary, rebooted, you'll want to quickly make sure your new device can be found, and set its ID. If you are going to connect over Ethernet, we strongly suggest you perform this step over USB first:

1) Plug in your new device into a USB port.

   For maximum capabilities, we recommend plugging into a USB port directly on the computer or to a powered USB hub. If you plug into an unpowered USB hub, you may run into sourcing limitations. This can also occur if you have multiple devices plugged into the hub, even if it is powered.

2) Go to Start – Programs – LabJack – LJControlPanel.

   The complete documentation for this program is installed with the LabJack driver, but there a few more steps to perform before starting DAQFactory.



3) Click on "Find Devices".

   This will search and hopefully find your new device. If your device is not found, you should refer to the LabJack User's Guide for more assistance.

4) Once found, click on the device.

   This will display current information about the device and an option to run a test panel. You may want to click on the test panel just to see that you are receiving signals

5) Set the ID: you should set the ID of your device to a non-zero value.

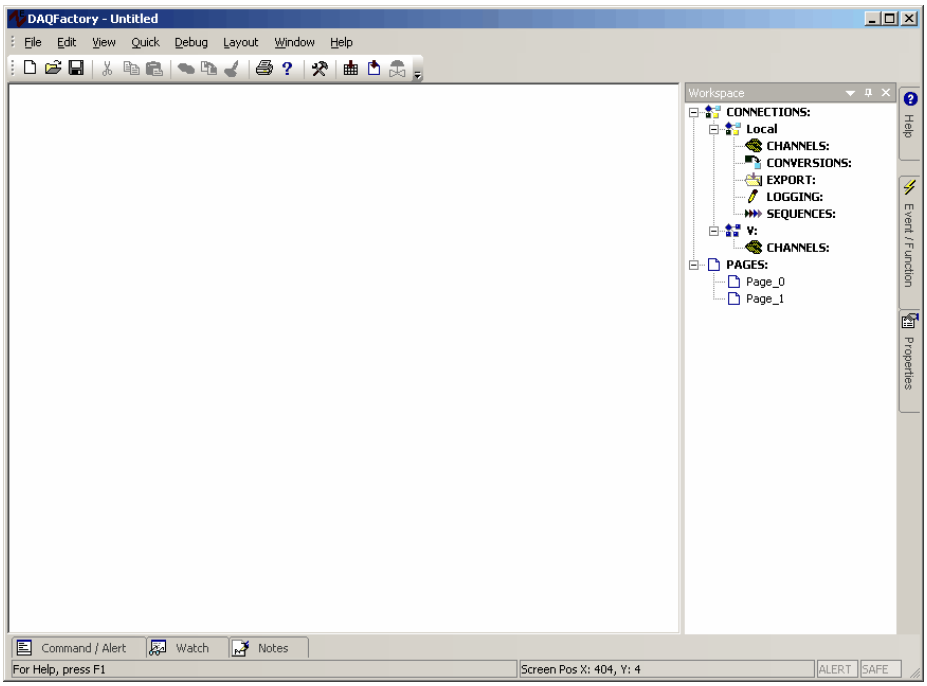
   If you are using multiple LabJacks, the ID will need to be unique. Zero is used by DAQFactory to indicate "First Found" and so cannot be used as LabJack ID. If you are using Ethernet, the ID is not required, but we still recommend using unique non-zero ID's just in case you plug it in to the USB port.

6) Save your settings and close the application.

2.3 Starting DAQFactory

Now you can start DAQFactory and start taking some data:

From your Start menu, select Programs => DAQFactory => DAQFactory. A splash window will appear, followed by the application:



The DAQFactory application is setup like most windows applications with a title bar at the top, a menu underneath, a toolbar underneath that and a status bar at the bottom. In addition there are a number of docking windows. A docking window means a window that can be "docked" or attached to the one side of the main application window. At this point, there are three important docking windows. Along the right is the Workspace. The Workspace is a listing of various elements of DAQFactory you will work with, such as channels, scripts, logging sets and pages. Also along the right, but initially tabbed, is the help window. Click on the Help tab to display the help window.

Along the bottom is the Command / Alert window. Normally it is rolled up, but if you click on the Command / Alert tab, it will appear. To have it stay visible, click on the thumb tack at the top right corner of the window.

The Command / Alert window allows you to manually type in commands. More importantly for the new user, this area displays any error messages that DAQFactory or your LabJack device might generate. Unlike most applications,

DAQFactory does not popup error message windows. This is because there are many situations where you may get a large number of continuous errors. For example, if you accidentally unplug your device while taking data. Instead of generating a huge number of windows with repetitive messages that you have to dismiss, DAQFactory puts all the messages into the Command / Alert window. This works well, but can sometimes be confusing, as the error message will remain in the Command / Alert window until it gets scrolled off the screen. To ensure that you don't confuse an old error message for a new one, you can right click on the command / alert window and clear the display.

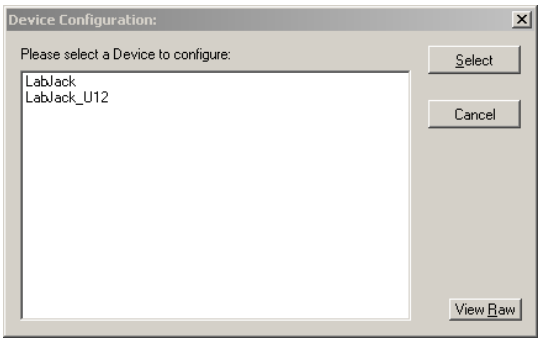
If for some reason you can't see any of these windows, go to the View menu and select the desired window.

Note: we strongly recommend keeping the Command / Alert window visible when you are starting, even though it takes up screen space. If you hide it, you may miss important alerts. The window will flash when an alert occurs, but this is not as apparent as a real message.

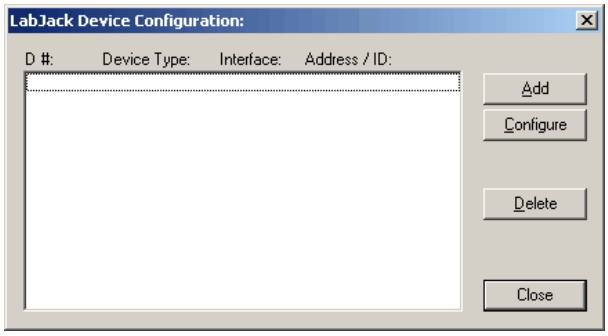
2.4 Ethernet setup

The LabJack UE9 offers both USB and Ethernet connectivity. So far, we have discussed communicating with LabJacks over USB. To connect to an Ethernet LabJack requires an extra few simple steps as the D# column of a DAQFactory channel only accepts numbers and not IP addresses:

1) Click on **Quick - Device Configuration** from the DAQFactory main menu. A new window will popup.



2) Select **LabJack** from the list of available devices. A new window will appear.



3) Click on the Add button to add a new LabJack device.

4) In the new window, select a unique device number to assign to this device.

This number should be unique among any other LabJack devices, even USB ones, as this is the number DAQFactory uses to identify the device.

5) Select **UE9** and **Ethernet** interface, and put the IP address of your device in the Address space.

6) Click **OK**.

The new device will be listed in the window. Depending on your DAQFactory installation, you may see a "Configuration" button. We do not recommend using this button to configure your LabJack, and it is considered depreciated. Instead, use the techniques described later in this guide.

7) Click **OK** to close this window.

At this point, the LabJack with the given IP address has now been assigned a device number that you can use everywhere you'd use the LabJack ID in DAQFactory. We have mapped the IP address to a device number. Do not use the LabJack's ID, but the device number instead.

3 Basic I/O, read/write analog/digital

3 Basic I/O, read/write analog/digital

3.1 Reading inputs

Performing basic analog and digital input and output with DAQFactory and a LabJack is quite simple. This is done by creating a channel for each desired I/O point.

Analog Input:

To create an analog input channel:

1) Click on the word **CHANNELS:** in the Workspace under **Local**. This will bring up the channel table.

Unless otherwise specified, when we say click on **CHANNELS:** we mean the one under **Local** and not under **V:**.

2) Click on **Add** to add a new row to the table

3) In the **Channel Name** column, put a name, say **`Pressure`**.

The channel name can be anything you'd like as long as it starts with a letter and contains only letters, numbers and the underscore.

4) In the **Device Type** column, select **`LabJack`**.

All your LabJack channels will use this Device. Make sure not to select LabJack_U12 as this is for the U12 only.

5) In the **D#** column, enter the ID of your LabJack.

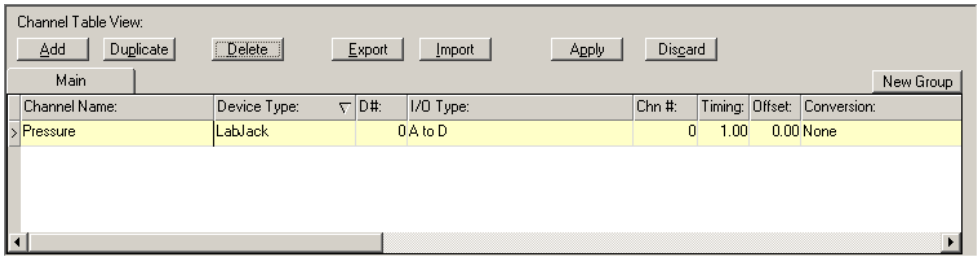
If you don't know the ID of your LabJack, please read the section on setting up your device in the last chapter. If you only have one LabJack and never expect to use any more, you can use 0 (zero) to have DAQFactory use the first found LabJack on USB. Please see the Ethernet setup section if you are using an Ethernet connection to your LabJack (UE9 only).

6) In the **I/O Type** column, select **`A to D`**.

7) In the **Channel** column, enter **`0`** to read the first analog input channel on your device.

8) The next column, **Timing**, determines how often the LabJack is queried for a new value in seconds. The default is once a second, and we can leave it at this setting.

9) Leave all the other settings as is and click **Apply**. As soon as you click Apply, acquisition will start.



10) To quickly see your data coming in, click on the **+** next to **CHANNELS:** in the workspace to expand the tree, then click on your **Pressure** channel to display the channel view for this channel.

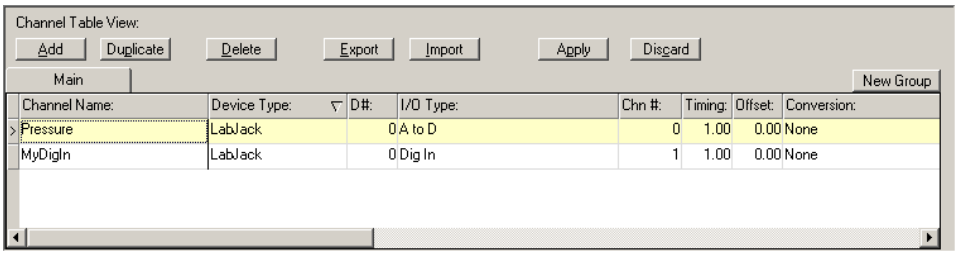
11) The channel view has multiple tabs. The first two repeat the settings in the channel table. Click on the last tab, **Table** to see your values coming in from the device.

This is not the primary way to view your data, but provides a quick indication that you are getting data. If you have nothing wired into your LabJack, you are most likely going to see a non-zero floating voltage reading.

On some LabJack devices, the pins can be either analog or digital input or output. DAQFactory automatically configures it to the appropriate setting, in this case an analog input.

To create a digital input, simply repeat the same procedure, using a different Channel Name, and selecting **`Dig In`**

for the I/O Type. If you are using the U3, which uses the same pins for analog and digital signals, select a different Channel Number than the one we used for analog (zero) to use a different pin. If using the U3-HV, you will have to choose a Channel number that is greater than 3 as the first 4 pins (0-3) are analog input only.



Sample file: ***LJGuideSamples\InputsSimple.ctl***

3.2 Differential analog inputs

On the U3, you can perform differential inputs assigning any other input channel as the negative input. To do so, put the channel number for the negative side of the input in the **Quick Note / Special / OPC** column. For the special 0-3.6 volt range, put channel number 32 in the Quick Note / Special / OPC column.

Sample file: ***LJGuideSamples\InputsSimpleU3Diff.ctl***

3.3 Channel Pinouts

DAQFactory uses sequential channel numbers. The various LabJack devices group their I/O pins. Here is how they correspond:

U3:

The U3 uses flex pins, meaning a pin can be either an analog input, digital input or digital output. They can also be counters or timers depending on configuration. For analog input, and digital I/O the pin mapping is:

I/O Pin:	Ch:	I/O Pin:	Ch:	I/O Pin:	Ch:
FIO0 (AIN0)	0	EIO0 (AIN8)	8	CIO0	16
FIO1 (AIN1)	1	EIO1 (AIN9)	9	CIO1	17
FIO2 (AIN2)	2	EIO2 (AIN10)	10	CIO2	18
FIO3 (AIN3)	3	EIO3 (AIN11)	11	CIO3	19
FIO4 (AIN4)	4	EIO4 (AIN12)	12		
FIO5 (AIN5)	5	EIO5 (AIN13)	13		
FIO6 (AIN6)	6	EIO6 (AIN14)	14		
FIO7 (AIN7)	7	EIO7 (AIN15)	15		

Note that the CIO pins are digital only. You should only use a pin for one of the three uses, so be careful not to use the same channel number for separate analog input, digital input or digital output channels.

If you enable timers or counters, they will use up the appropriate number of pins starting with FIO0. This does not change the rest of the channel numbers. So, if you have two timers enabled, they will be on FIO0 and FIO1. FIO2 remains channel 2 to DAQFactory. Of course if you use the pin offset option for timers and counters, the timers

might be on different pins, but even the mapping of DAQFactory channel numbers to pins remains the same.

UE9:

The UE9 has separate pins for analog and digital I/O. The analog input pins correspond directly to channel numbers, so AIN0 is channel 0, AIN1 is channel1, etc. For digital I/O, the mapping is:

Channel:	**I/O pin:**
0-7	FIO0-FIO7
8-15	EIO0-EIO7
16-19	CIO0-CIO3
20-22	MIO0-MIO2

If you enable timers or counters, they will use up the appropriate number of pins starting with FIO0. This does not change the rest of the channel numbers. So, if you have two timers enabled, they will be on FIO0 and FIO1. FIO2 remains channel 2 to DAQFactory. Of course if you use the pin offset option for timers and counters, the timers might be on different pins, but even the mapping of DAQFactory channel numbers to pins remains the same.

3.4 Setting outputs

Creating output channels is identical to creating input channels with one important difference. Since they are output, there is no Timing parameter. In fact, if you select Analog Output or Digital Output as an I/O type, you will not be able to edit the channel's timing parameter. So, to create an output channel, we follow the same steps as an input:

1) Click on the **CHANNELS:** in the Workspace under **Local**.

   This will bring up the channel table.

2) Click on **Add** to add a new row to the table.

3) In the **Channel Name** column, put a name, say **`Output`**.

   The channel name can be anything you'd like as long as it starts with a letter and contains only letters, numbers and the underscore.

4) In the **Device Type** column, select **`LabJack`**.

   All your LabJack channels will use this Device. Make sure not to select LabJack_U12 as this is for the U12 only.

5) In the **D#** column, enter the ID of your LabJack.

   If you don't know the ID of your LabJack, please read the section on setting up your device in the last chapter. If you only have one LabJack and never expect to use any more, you can use 0 (zero) to have DAQFactory use the first found LabJack on USB. Please see the Ethernet setup section if you are using an Ethernet connection to your LabJack (UE9 only).

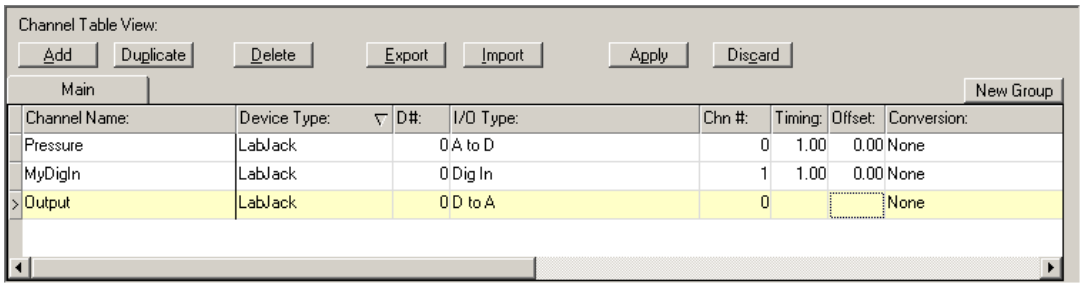
6) In the **I/O Type** column, select **`D to A`**.

7) In the **Channel** column, select **`0`** to set the first analog output channel on your device.

8) As mentioned, the next column, **Timing**, is not used for outputs.

9) Leave all the other settings as is and click **Apply**.

   Unlike inputs where acquisition starts immediately on Apply, creating an output channel does not actually set it to a value.



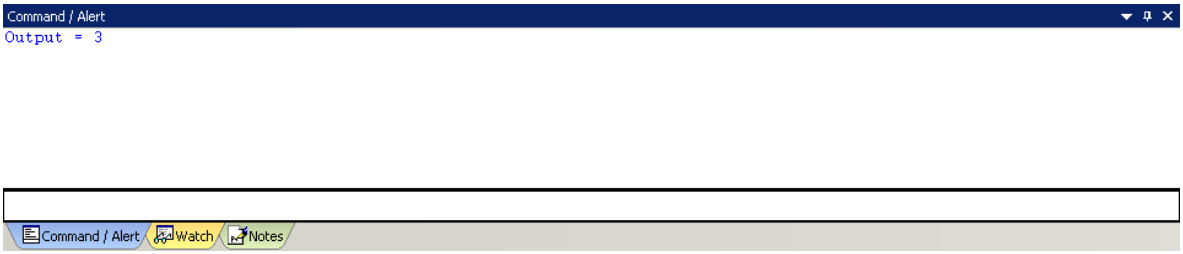
10) To quickly set the output to a value we can use the Command / Alert window. This is not the primary method of setting an output, but a quick way to test your settings. We'll show a better way in the next chapter. The Command / Alert window may be rolled up at the bottom of the screen. Click on the Command / Alert tab to display it:

If you can't see the Command / Alert tab, try going to View - Command / Alert

The Command / Alert window is made up of two parts, the status / alert display, which takes up most of the window, and the command line, which is at the bottom and takes up one line. You can type in one line commands here and see the results in the status display. You can also do commands that have no display, but change things, such as outputs. So, to set our new channel to a value of 3 volts, we type:

```
Output = 3
```

and hit Enter. There will be no confirmation other than the command itself being displayed in the status / alert area, and no error displayed. Your output will also be at 3 volts.



Like the input channels, DAQFactory will automatically set your device into the proper settings for pins with multiple duties. So, for digital pins, for example, DAQFactory will automatically set them into output mode.

Sample file: ***LJGuideSamples\InputsSimpleWithDA.ctl***

4 Display the data

4 Display the data

4.1 Manipulating screen components

Before we explain how to display your data, we should quickly explain how to manipulate screen components in DAQFactory. Because DAQFactory is a dynamic application, meaning you can make edits while your application is actually running, DAQFactory does not use the typical Windows click-and-drag style screen component manipulation. Instead, you must hold down the Ctrl key. While the Ctrl key is pressed, you can click on components to select them, click-and-drag to move them around, and click-and-drag their corners to resize (for resizable components).

If you don't like this, you can put the system into edit mode by selecting Edit - Edit Mode from the DAQFactory main menu. This will reverse the functionality of the control key when working with components (except keyboard nudging). You can simply click and drag components, but if you want to click on a button to do an action you will have to hold the Ctrl key down. We strongly recommend against using Edit Mode except for when you have a lot of repetitive edits to make on a large number of components.

For those that skim and didn't read the previous paragraphs:

In order to manipulate screen components, you must hold down the Ctrl key.

4.2 Variable value components for numeric display

The Channel View certainly provides a quick look at your data, but is not the most convenient or flexible way to display your data, especially once you have more than one channel. In this section we'll add a component to a page that will display data from the Pressure channel. This assumes, of course, that you created an analog Pressure channel in the basic I/O chapter. If you didn't, load the document *LJGuideSamples\InputsSimpleWithDA.ctl* and start from there.

1) Click on **Page_0** in the Workspace under **PAGES:**.

   This will display a blank white page unless you are working off InputsSimpleWithDA. When DAQFactory Express starts, 2 blank pages are generated automatically with default names. With Express you are limited to 2 pages, but with all the other versions of DAQFactory you can create as many pages as you need.

2) Right-click somewhere in the middle of the blank area and select **Displays** - **Variable Value**.

   This will place a new page component on the screen. There are many different components available that allow you to create custom screens. This particular component simply displays a changing value in text form.

3) Right click on the new component and select **Properties...**

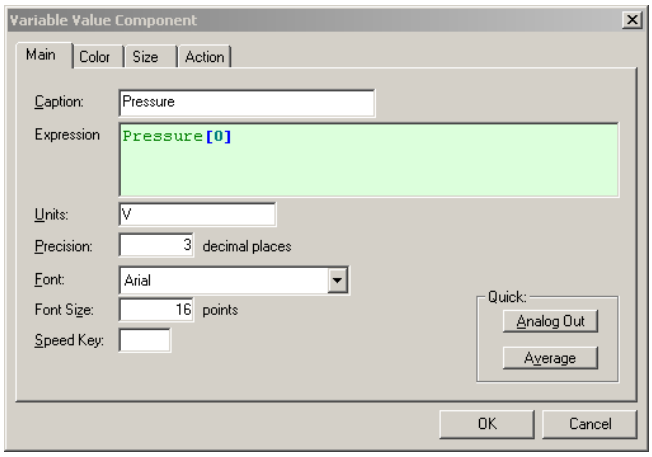
   All of the screen components have different properties windows for configuring the look of the component and telling the component what data to display.

4) Next to **Caption:** enter **`Pressure`**.

   The caption is what is displayed in front of your value. A colon is automatically added when displayed.

5) Next to **Expression:** enter **`Pressure[0]`**.

   Expressions are actually formulas that you can enter to display calculated data. In this case, the formula simply requests the most recent data point of the channel pressure. The [0] indicates the most recent data point. If, for example, we wanted to convert the pressure voltage into actually pressure units, we could put something like Pressure[0] / 5 * 500.



6) Click **OK**.

Now the screen should display Pressure: 0.42 V with a changing number. This is the current value in the pressure channel we created earlier. Feel free to open the properties window again and play with the units, font and other settings. When you are done, load the sample file listed below as we'll build on it in the next section.

Sample file: ***LJGuideSamples\VariableValue.ctl***

4.3 Conversions vs direct expressions

In the previous section we displayed the voltage reading from our analog input channel. However, in most cases you really want to view your data in more appropriate units than volts. With pressure, for example, you might want PSI or millibar. There are two ways you can convert your data into different units.

The first way is to simply put the desired formula in the Expression area of the component properties. So, if you had a 0-5V pressure sensor with a range of 0-500 psi, you would put:

```
Pressure[0] * 100
```

to convert your voltage signal to PSI. You would also, of course, probably want to change the unit display. You can put much more complicated calculations as well, even ones based on multiple channels. DAQFactory supports a wide variety of operators and mathematical functions to make this easy. They are all listed in the Expressions chapter of the DAQFactory help.

Entering formulas directly into the Expression area of a component is handy because you can create another component to display the same input reading in different units. If, however, you always want your reading in particular units, you can instead use Conversions. This has the added benefit of allowing you to log your data in these units. Using our pressure example and a 0-5V : 0-500psi sensor, here is how you would create a conversion to put the readings into PSI:

1) Click on **CONVERSIONS:** in the Workspace.

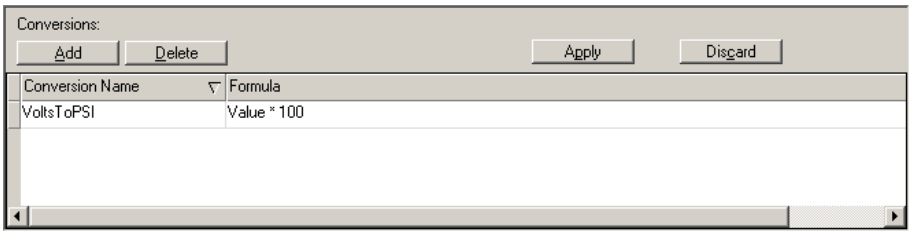
This will display the conversion table.

2) Click **Add** to add a new row.

3) Give the conversion a name, say **`VoltsToPSI`**

4) In the **Expression** column put:

```
Value * 100
```

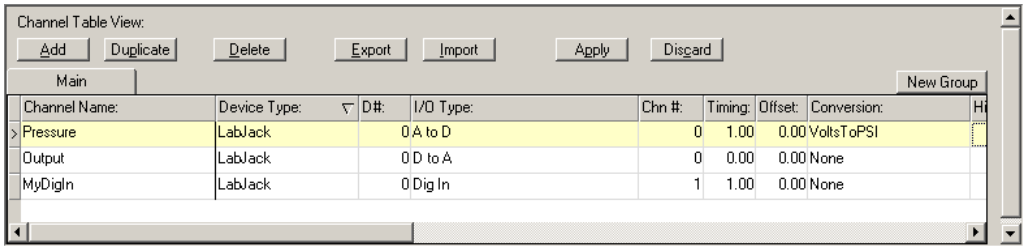


You'll notice that we used the word **`Value`** instead of the channel name in the conversion's expression. This allows a single conversion to be used on multiple channels. The actual channel reading is substituted wherever **`Value`** appears in the formula.

5) Click **Apply** to save the new conversion.

6) Click on **CHANNELS:** in the Workspace to bring up your list of channels.

7) In the row with you Pressure channel, go over to the column labeled **Conversion** and select **`VoltsToPSI`**.



8) Click **Apply** to save your changes.

If you then go back to Page_0 you will see that your values are now 100 times their previous size.

Note: just to reinforce this: **`Value`** is a keyword and not the name of a channel or variable. When you apply the conversion to the channel, that channel's current reading is substituted where **`Value`** is found in the conversion.

Sample file: ***LJGuideSamples\Conversions.ctl***

4.4 Descriptive text components for textual display

The variable value component is great for analog data, but less so for digital data where the only values are 0 and 1. For this, the Descriptive Text Component is useful to allow you to display a word or message for 0 and 1 instead.

To show this, let's use the "Output" D to A channel we created in the last chapter and pretend it is a digital input channel.

1) On a page, right click and select **Displays** - **Descriptive Text**.

2) Once the new component is on the screen, right click on it and select **Properties...**.

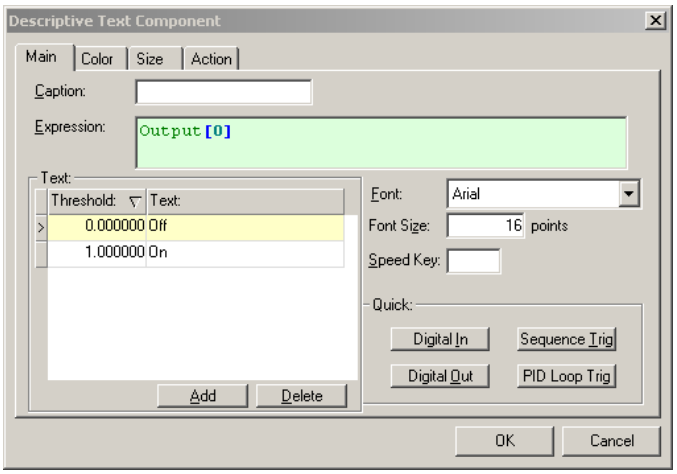
3) For the **Expression** type: **`Output[0]`**

4) By default, the component already will display the word "Text". Double-click on the word yellow highlighted word Text in the table at the bottom left of the properties to select it and replace it with the word **`Off`**.

This will cause the component to display the word Off when Output is 0.

5) Next, click on the **Add** button to create a new row in the table, and put **`1`** for the Threshold and **`On`** for the Text.

This will cause the component to display the word On when Output is 1. It doesn't matter what the order of the rows are in the table. DAQFactory will sort them once you close the window.



6) Click **OK**. Then go to the command / alert window and type **`Output = 0`** and hit Enter and watch the new component display Off. Type **`Output = 1`** and hit Enter and the component will change to On.

DAQFactory actually provides a quicker way to set this up.

7) Right click on our text component and select **Properties…**.

8) Click on the button towards the bottom right labeled **Digital In**. This will popup a window requesting a channel. Type **`Output`** and click **OK**. Then click **OK** to close the properties.

> This button essentially did steps 3 through 5 in one step. It also setup the colors so that Off displays in red, and On displays in green. Go ahead and type Output = 0 and Output = 1 in the Command / Alert window to watch it change.

As you may have noticed, there is also a button labeled **Digital Out**, which does almost the exact same thing as the Digital In button, except that it makes the descriptive text component into a clickable component that will toggle the output between 0 and 1. It does this by changing the component's Action. You can see this by clicking on the **Action** tab of the components properties. Feel free to create a new Descriptive Text component then go into the component's properties, click **Digital Out**, enter **`Output`**, then on closing simply click on the component to toggle Output instead of going to the command alert window.

Sample file: ***LJGuideSamples\DescriptiveText.ctl***

4.5 Graphing

Displaying scalar values in text form is certainly useful, but nothing is better than a graph to give you a handle on what is happening with your system. DAQFactory offers many advanced graphing capabilities to help you better understand your system. In this section we will make a simple Y vs time or trend graph.

1) If you are still displaying the page with the value on it, hit the 1 key to switch to Page_1. If you are in a different view, click on **Page_1** in the workspace.

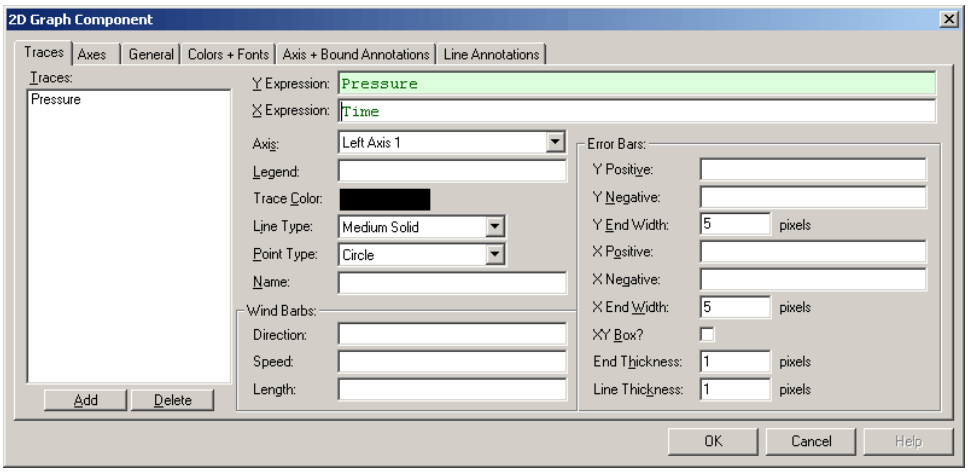
> The workspace is only one way to switch among pages. Another is using speed keys which can be assigned to a page by right clicking on the page name in the workspace and selecting **Page Properties…**.

2) Right click somewhere on the blank page and select **Graphs** - **2D Graph**. Move and resize the graph so it takes up most of the screen by holding down the Ctrl key and dragging the component. Drag the small black square boxes at the corners of the selected graph to resize. Next, open the properties window for the graph by right clicking on the graph and selecting **Properties…**.

3) Next to **Y Expression:** type **`Pressure`**.

> The Y expression is an expression just like the others we have seen so far. The difference is that a graph expects a list (or array) of values to plot, where in the previous components we have only wanted the most recent value. By simply naming the channel in the Y expression and not putting a [0] after it, we are telling the

graph we want the entire history of values stored in memory for the channel. The history length, which determines how many values are kept in memory and therefore how far back in time we can plot is one of the parameters of each channel that we left in its default setting of 3600.



4) Leave all the rest in their default settings and click **OK**.

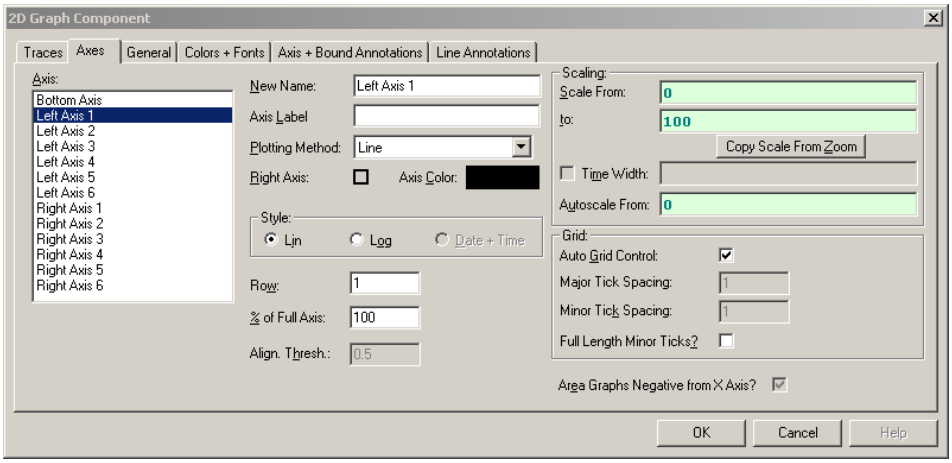
You should now see the graph moving. The data will move from right to left as more data is acquired for the pressure channel. However, we currently are multiplying our Pressure channel reading by 100 using the Conversion we created a few sections ago, so the trace is completely off scale and not visible. To display the data, we need to change the Y axis scaling.

5) Double click on the left axis of the graph.

Unlike other components, you can double click on different areas of the graph to open the properties window for the graph to different pages. Double clicking on the left axis brings up the axis page with the left axis selected. You can also just open the properties window for the graph and select the **Axes** tab, then **Left Axis 1**.

6) Next to **Scale From** enter **`0`**, next to **Scale To** enter **`100`**, then click **OK**.

This will scale the graph from 0 to 100. Depending on what you have plugged into your LabJack input, or what voltage the LabJack is floating at, this may or may not be a big enough scale. You can always click on **Page_0** in the workspace to see what the current Pressure reading is and change the scaling of the graph appropriately.



Now we'll try another method to scale the graph.

7) Open the graph properties box again and go to the **Traces** page. Change the **Y Expression** to **`Pressure / 100`** and hit **OK**.

Like the other screen components, the expressions in graphs can be calculations as well as simple channels. In this case, we are dividing each of the values of Pressure's history by 100 and plotting them. This is essentially

removing the conversion we applied to Pressure in the previous section since it multiplied our reading by 100.

8) Deselect the graph by clicking on the page outside of the graph. The shaded rectangle around the graph should disappear. Next, right click on the graph and select **AutoScale** - **Y Axis**.

The graph component has two different right click popup menus. When selected, the graph displays the same popup menu as the rest of the components. When it is unselected, it displays a completely different popup for manipulating the special features of the graph.

After autoscaling, you should see the graph properly scaled to display your signal zoomed in. Notice the rectangle around the graph. The left and right sides are green, while the top and bottom are purple. The purple lines indicate that the Y axis of the graph is "frozen". A frozen axis ignores the scaling parameters set in the properties window (like we did in step 6 above) and uses the scaling from an autoscale, pan, or zoom.

9) To "thaw" the frozen axis, right click on the graph and select **Freeze/Thaw** - **Thaw Y Axis**

Once thawed, the graph will revert to the 0 to 100 scaling indicated in the properties box and the box surrounding the graph will be drawn entirely in green. At this point you may wish to change the scaling to 0 to 5 using steps 5 and 6 above since we have divided Pressure by 100.

10) Double click on the bottom axis to open the properties to the axis page with the bottom axis selected. Next to **Time Width**:, enter **`120`** and click **OK**

If the double click method does not work, you can always open the properties window for the graph using normal methods, select the **Axes** page and click on **Bottom Axis**.

In a vs. time graph, the bottom axis does not have a fixed scale. It changes as new data comes in so that the new data always appears at the very right of the graph. The time width parameter determines how far back in time from the most recent data point is displayed. By changing it from the default 60 to 120 we have told the graph to plot the last 2 minutes of data.

Once again, if you zoom, autoscale, or pan the graph along the x axis, it will freeze the x axis and the graph will no longer update with newly acquired values. You must then thaw the axis to have the graph display the trace as normal.

Sample file: ***LJGuideSamples\Graphing.ctl***

4.6 Outputting with variable value component

In addition to taking data, most systems also control outputs. Here we will create an output channel and control it manually from a page component. We'll assume you still have that Output channel we created earlier.

1) Click on **Page_0** in the Workspace to go to our first page.

2) Right-click in a blank area of the page and select **Displays**-**Variable Value** to create another variable value component.

3) Right click on the new component and select **Properties...** to open the properties window.

4) For the expression, enter **`Output[0]`**

Like before, this will simply display the most recent value of the out channel. Feel free to set the caption as well.

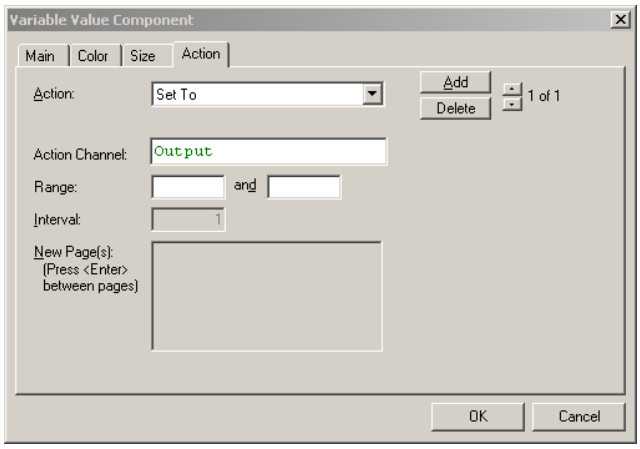
5) Click on **Action** tab.

This tab exists on several different components including the static ones and works the same way with all of them.

6) From the **Action** drop down, select **Set To**

There are many different options for the Action described in the DAQFactory help. The Set To action will prompt the user for a new value when the component is clicked and apply the entered value to the channel or variable.

7) Next go to **Action Channel:** type **`Output`**



8) Leave the **Range** blank and click **OK**.

The range allows you to constrain the inputs to particular values. By leaving these properties blank, we are indicating that we do not want a range limitation. The page will now display your caption and either the most recent setting for Output, or a 0 with a big red X through it if you started from the last sample document. The red X indicates that Output does not have a valid value yet. This is because we haven't set it to anything.

9) Click on the component. A new window will appear requesting a new value. Enter a voltage value within the DAC range, such as **2**, and click **OK**.

Output will now be set to the value you entered. The component will now display your new value without the big red X. If you tie a wire between DAC0 and FIO0 (U3) or AIN0 (UE9) you will see the Pressure reading track with the DAC output, though multiplied by 100.

Note, if you'd like to be able to enter an output value in units other then volts, you need to create a Conversion and apply it to your output channel. We talked about conversions a little earlier. Conversions on output channels work in reverse. With regular input conversions, the formula is used to convert the devices units, usually volts, to your units. With output conversions, the formula is used to convert your units into the device units. So, if you had a variable pressure controller that controlled from 0 to 500 psi and took a 0 to 5V output, the conversion would be:

```
Value / 100
```

*Sample file: **LJGuideSamples\AnalogOut.ctl***

4.7 Toggling a digital output using a button

We talked a little about using a descriptive text component to toggle a digital output between its two states at the end of section 4.4. You can also create a button to toggle the output:

1) On a page, right click and select **Buttons & Switches** - **Button**.

2) Once the new component is on the screen, right click on it and select **Properties…**.

3) For the **Caption** give whatever label you want

4) Click on the **Action** tab.

5) For **Action**, select **Toggle Between** from the drop down combobox.

6) For **Action Channel** type in the name of your digital output channel.

7) For **Toggle Between**, put 0 and 1. For digital outputs, only 0 and 1 apply, but you could also use this for toggling an analog output between two voltages by specifying other values here.

8) Click **OK**

Now you can click on the button and your digital output channel will toggle between 0 and 1.

*Sample file: **LJGuideSamples\DescriptiveText.ctl***

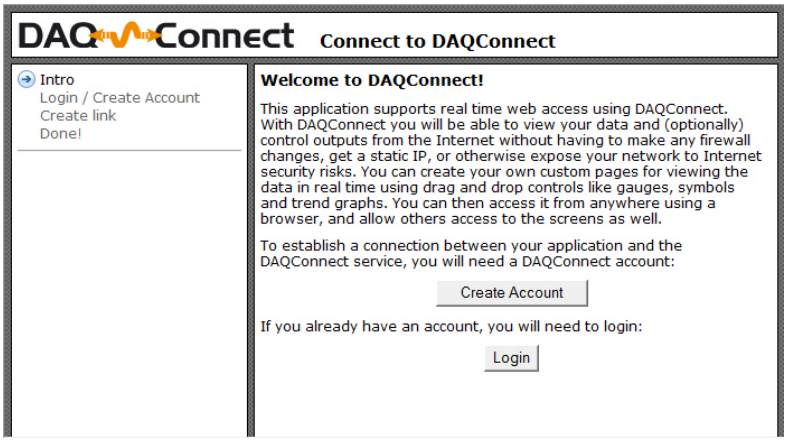
4.8 Viewing Data on the Web

Once you have DAQFactory taking some data, you can also view it on the web in a browser using the DAQConnect service. To do this, we'll create a free DAQConnect account, setup our Pressure channel to send its measurements to DAQConnect, and create some screens in a browser.

DAQConnect allows you to view data live from a web browser and optionally do remote control without having to worry about firewall or router settings, static IPs or any other IT issues. You also do not need to have your own website, just a DAQConnect account. You can create screens in DAQConnect directly in your browser using drag and drop controls, and can view these screens from anywhere, even your iPhone, iPad or other mobile device.

1. Click on **Real-time Web -> Connect...** from the main menu.

This will popup a window that allows you to create a DAQConnect account and create or select a DAQConnect data source to send your data to. A data source is a collection of channels, typically from one location within your DAQConnect account. You can have multiple data sources within your account if you have multiple locations or you otherwise want to separate your data into groups.

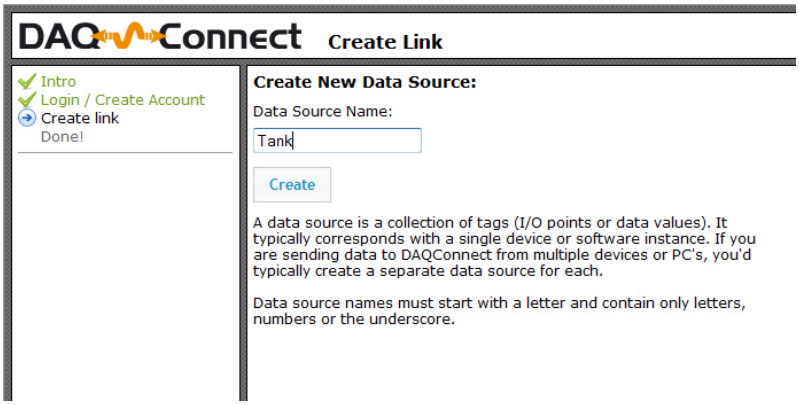


2. Click on **Create Account** when the DAQConnect window opens, or if you already have a DAQConnect account, click **Login**, login and proceed to step 4.

This will create a DAQConnect account, allowing you to send data to the web. There's a free plan so you can try it without providing a credit card or much of anything else. You will not need to login every time you start DAQFactory, just when you want to link a DAQFactory application to a DAQConnect data source.

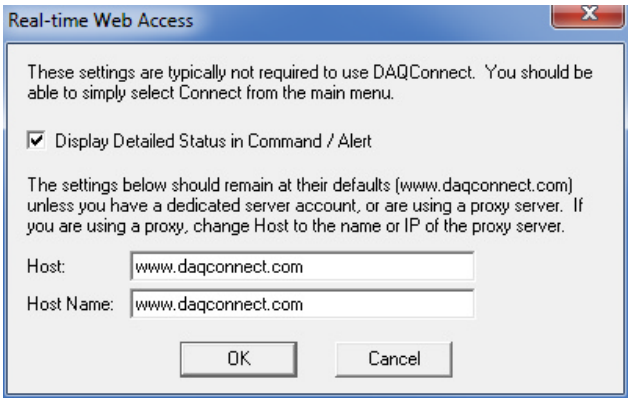
3. Fill in the form with the appropriate details and click **Create Account**. When the Select Plan page appears, you can leave it on the free plan, or select a higher plan for more capabilities and click **Continue**. If you select a paid plan, you will be taken to the credit card page, otherwise you will go directly to the screen to create a new data source.

4. Enter **Tank** for your new data source name and click **Create**.



This will create a new data source named "Tank" and link it to your DAQFactory document. You can create other data sources in your account and switch among them by clicking on Real-time Web -> Connect again.

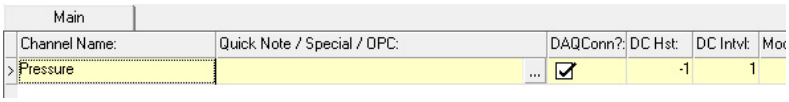
5. The window will close. Select **Real-time Web -> Settings**. Then check **Display Detailed Status in Command / Alert** and click **OK**.



This will cause DAQConnect status messages to appear in the command / alert window. This setting is designed for debugging only, and is not saved with the document.

If all is well, you should see DAQConnect status messages, probably saying "Sending: 0 tags to server" in the Command / Alert window located at the bottom of the DAQFactory window. Now that your DAQFactory document is connected to DAQConnect, we can have our Pressure channel send its data to DAQConnect.

6. Click on **CHANNELS:** in the workspace, find the **Pressure** channel, and check the box under the column labeled **DAQConn?**. Click **Apply**.

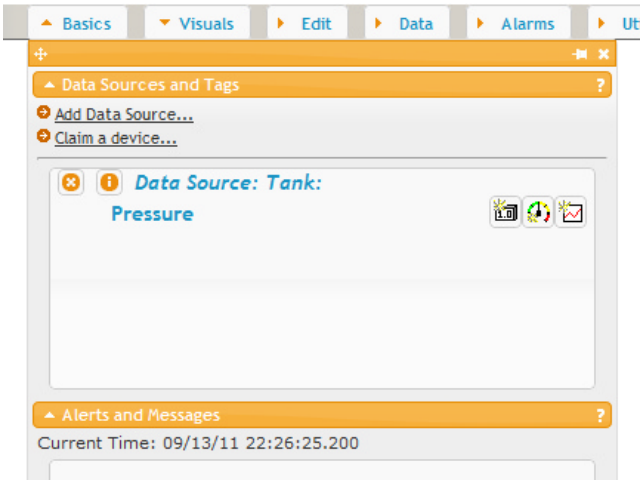


The pressure channel will immediately start sending its data to your DAQConnect data source. Now we can go to DAQConnect in a browser and view the data:

7. Open up your favorite browser and browse to www.daqconnect.com then click on **customer login** or **my account** if you are already logged in. Feel free to do this on a different computer. Login using the account you created in step 2 above.

Although you can use pretty much any browser, we recommend Chrome because it has the fastest JavaScript implementation, then Firefox, and finally, IE, which unfortunately still falls way behind in its JavaScript performance.

8. Once logged in, you will be presented with the main DAQConnect editor and a blank page. Click on the **Basics** menu at the top. You then see your Tank data source listed with the Pressure channel (or tag as its called in DAQConnect) listed underneath. To the right of Pressure are three icons. Drag the first one onto the big white area (the page) outside the menu to view the current Pressure reading as a number.



9. Try dragging the other two icons to the page to view a gauge and a trend graph of your data.

There are a number of other controls available in the Visuals menu. See the DAQConnect help for more information.

10. Click on **Save** at the top right corner to save your new DAQConnect page. Now you can login from anywhere to view real time values!

There are ways to create restricted, read-only members with separate login for the same account, as well as the ability to embed your dynamic DAQConnect pages inside other web pages. Please refer to the DAQConnect help for more information.

That is the very basics of DAQConnect. Feel free to explore more with your free account, and refer to the DAQConnect help for more details. There is also more detail on using DAQFactory with DAQConnect and control capability in the DAQFactory help.

5 Logging

5 Logging

DAQFactory offers a number of different logging options. There are logging sets, export sets, and for those that need total flexibility, the File. series of functions for direct file access.

Logging sets are used to do continuous logging. They can be started and stopped, but are not designed for conditional logging because of the way they collect their data. Typically you will run a logging set in one of two modes:

All Data Points (aligned): this will log all data from any channel you assign to the logging set. If your data is coming in with different time stamps, it will try and align the data points to put them on the same line in the log based on the align parameter. The default align parameter of 0 means that if two data points have a different time, they will end up on a separate line in the log. You can increase this and data will be combined on the same line if their times are closer than the align threshold (in seconds). The only problem is that if your data is coming in faster than the align threshold on a single channel, some of that channel's data will not get logged.

Fixed Interval: this will cause the logging set to go out every Interval (in seconds) and either average all the data accumulated on each channel and log the average, or take a snapshot of whatever the most recent reading was. This is useful if, for example, you want to simply log your data every minute, even though some data points are coming in once a second, others every 5 seconds, etc. You'd select Fixed Interval, then Snapshot and an Interval of 60.

The File. functions are more advanced and covered in the DAQFactory user's guide.

5.1 Logging to ASCII files

Next we will learn how to store this data to disk so that it can be opened with another program for analysis. In this example we will create a simple comma delimited file with our pressure data.

1. Click on **LOGGING:** in the workspace.

   This will display the logging set summary view. Here you can see all your logging sets displayed and their running status. A logging set is a group of data values being logged in a particular way to a particular file. You can have as many logging sets as you need, and they can run concurrently.

2. Click on the **Add** button in the logging set and when prompted for the new name, type in **`MyLog`** and click **OK**.

   Like channels and all other DAQFactory names, the logging set name must start with a letter and only contain letters, numbers or the underscore. Once you click OK, the logging set view will be displayed for the new logging set. You can also get to this view by clicking on the logging set name listed under LOGGING: in the workspace. You may have to click on the + sign next to LOGGING to see your logging sets listed in the workspace.

3. Next to **Logging Method**, select **`ASCII Delimited`**, which is actually the default.

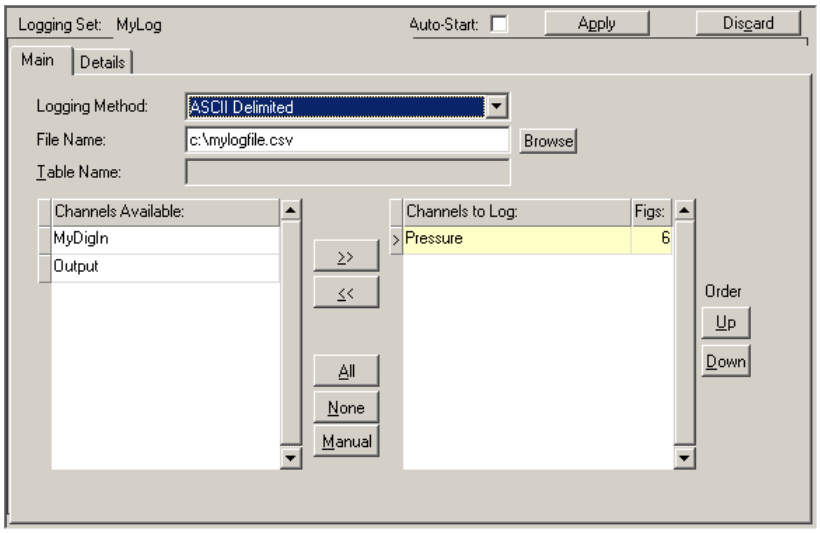
   ASCII Delimited is probably the most common method for data logging as it can be read by most other programs such as Excel. Unfortunately, it is not as space efficient or fast as the binary methods. But unless you have strict space constraints or you are logging more than about 50,000 points per second (depending also on your computer / hard drive speed), we suggest ASCII.

4. Next to **File Name** enter **`c:\daqfactory\mylogfile.csv`**

   It is usually best to fully specify the path to your file, otherwise the data will be stored in your DAQFactory directory. The **`.csv`** is a windows standard designation for comma delimited values, which, unless you change the delimiter on the details page, is the type of file that will be created.

5. In the **Channels Available** table, click on the row with **Pressure**, then click on the **>>** button to move it to the **Channels to Log** table.

   Each logging set can log any combination of channels. In this case, we will just log the input channel.



6. Click on **Apply** to save the changes.

7. To start logging, click on the **+** next to **LOGGING** to display the new logging set, then right click on the logging set **MyLog** and select **Begin Logging Set**.

Once started, the icon next to MyLog will change and the red stop sign will be removed to indicate that this logging set is running.

8. Wait at least 10 or 15 seconds to log some data and then right click on the logging set again and select **End Logging Set** to stop logging.

There are other ways to start and stop logging sets, including the Action page of some components such as the variable value component that we used earlier.

9. Now start Excel or Notepad and open the file c:\daqfactory\mylogfile.csv.

You will see two columns, time and data. By default the time is given in Excel / Access format which is decimal days since 1900. You can also have DAQFactory log in its standard format of seconds since 1970. If you are using Excel and logging the time in Excel format, you can format the first column as date / time and the numbers displayed will be properly displayed as date and time.

Hint: if you cannot find the mylogfile.csv file, check to make sure that you selected Pressure to log and not Output. Since Output is an output channel, it only gets new values to log when you actually set it to something.

Sample file: ***LJGuideSamples\LogASCII.ctl***

5.2 Batch logging

The method we just used can also be used for batch logging. Batch logging is used when you are doing multiple groups of measurements and you wish to log each group into a separate file. We already know how to create a logging set, so to do batch logging requires two more things:

a) a way to start and stop the batch

b) a way to change the file name, or, automatically assign the filename

Starting and stopping the batch log:

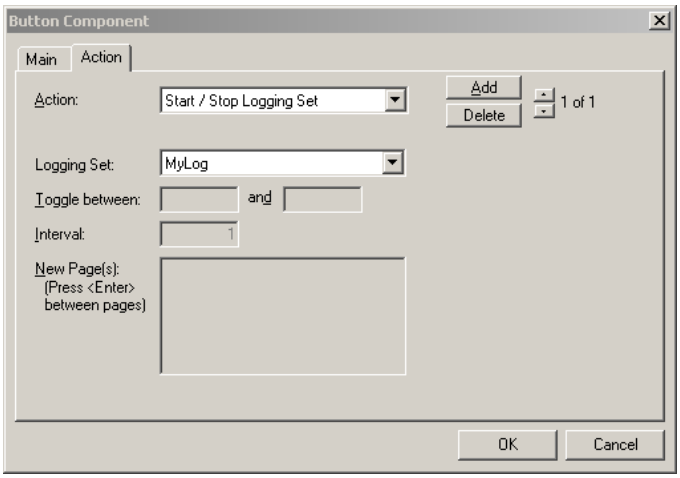
Really this is just starting and stopping a logging set. The easiest way is to simply use a button:

1) On a page, right click and select **Buttons**-**Button**.

2) Right click on your new button component and select **Properties…**. Change the caption to **`Start / Stop`**.

3) Click on the **Action** tab and select **Start / Stop Logging Set**, then in the combo, select our **`MyLog`** logging set, then

click **OK**.



At this point we have a button that will start and stop the logging set. But, it doesn't tell us if the logging set is actually running. For this, we'll use a descriptive text component. We can treat the status of the logging set, which is either running or not running, as a digital input:

4) Right click on the page again and select **Displays** - **Descriptive Text**.

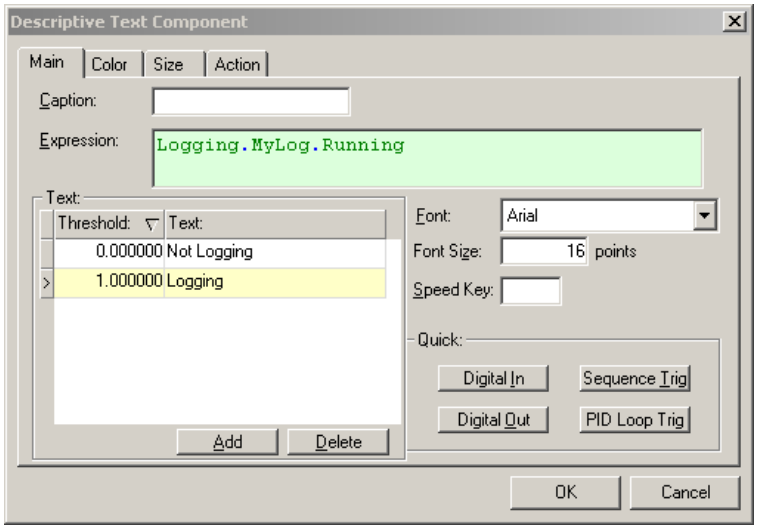
5) Right click on this new component and select **Properties…**.

6) For the Expression type:

`logging.mylog.running`

7) Change the word **Text** in the Text table to **`Not Logging`**, and then click **Add** to add a new row, putting **`1`** in the new row under **Threshold**, and **`Logging`** for the **Text**. Click **OK** to close the window.

The logging.mylog.running is a variable of the our "mylog" logging set which will equal 0 if the logging set is not logging, and 1 if it is. To differentiate between the "running" variable of the "mylog" logging set and the running variable of other logging sets, sequences, etc. we have to specify exactly what we want. "logging." means we want a logging set. "mylog." means we want the logging set called "mylog", and "running" is a variable of the logging set mylog.



Now, if you click on the button you will see the descriptive text component change from Not Logging to Logging and back. The logging set is also starting and stopping.

Changing the logging file name:

Now that we can start and stop the logging set, we need to create a way to change the file name of our logging set. The file name is just another variable of the logging set: "strFileName", so we'll access it using **logging.mylog.strFileName**:

1) Right click somewhere on the page and select **Displays** - **Variable Value**

2) Right click on the new component and select **Properties…**.

3) Change the **Caption** to **`Logging Path`**, set the **Expression** to: **`logging.mylog.strFileName`**, and clear out the **Units**. This sets up the display.

4) Click on the **Action** tab, and then select **`Set To`** from the available actions.

5) For Set To Channel, put **`logging.mylog.strFileName`**. Click **OK**.

At this point, you can click on the display of the logging path and it will prompt you for the file path. But, it'd be nicer if it displayed a standard windows File Save Window instead so the user could browse for the file:

6) Right click on our variable value component again and select **Properties…**.

7) Go back to the **Action** tab and select **`Quick Sequence`**

8) In the white space that appears, enter the following script:

```
private string path = File.FileSaveDialog(logging.mylog.strFileName)
if (!IsEmpty(path))
   logging.mylog.strFileName = path
endif
```

9) Click **OK** to close, then click on the path again to see it work.

To explain the script:

In the first line, we declare a string variable called **`path`**. It gets the result of the **`FileSaveDialog()`** function which is part of a group of functions which are always prefixed with **File.** . The **FileSaveDialog()** displays the standard Windows File Save window.

In the second line, we look to see if the **path** variable is empty. This would occur if the user hit **Cancel** from the Save window.

If the user didn't hit Cancel, then we get to the third line, which assigns the path they selected to the logging set.

Automatically setting the file name:

Finally, lets quickly demonstrate how to automatically set the file name based on the time and date. What we need to do is change the start / stop logging button to change the file name to include the current time whenever the logging set is started:

1) Right click on the **Start / Stop Logging** button we created at the beginning of this section, and select **Properties…**.

2) Go to the **Action** tab and change the **Action** to **`Quick Sequence`**. Enter the following script:

```
if (logging.mylog.running)
   endlogging(mylog)
else
   logging.mylog.strFileName = "c:\daqfactory\mydata_" + formatdatetime("%y%m%d_%H%M%S",systime()) + ".csv"
   beginlogging(mylog)
endif
```

3) Click **OK** and give the button a try. You should see the variable value component update with a new filename every time the logging set starts up. Note that clicking on the variable value component to change the filename has been rendered moot as the start / stop button will reset the filename.

To explain the script above:

In the first line we see if the logging set is running.

If it is running, then in the second line we stop the logging set.

Otherwise, in the fourth line, we set the filename. The filename is made up of three strings combined together. The first and last parts are static, "c:\mydata_" and ".csv". The middle part contains the date. The FormatDateTime() function takes what is called a specifier that determines how you would like the date and time displayed. In this case it will be Year Month Day_Hour Minute Second with no spaces. Please see the DAQFactory help for more details on possible specifiers. The second part of the FormatDateTime() function takes the time you want to format. In this case, we use SysTime() which is the current system time.

Sample file: ***LJGuideSamples\LogBatch.ctl***

5.3 Doing daily logs

Another form of batch logging is daily logging. This is when you create a new logging file every day (or week, hour, or whatever interval). This can easily be done with a simple sequence script. Assuming we still have the MyLog logging set:

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Call it **`DailyLog`**.

2) Once the sequence editor appears, check the box labeled **AutoStart**.

3) Enter the following script:

```
while(1)
   logging.log.strFileName = "c:\daqfactory\mydata_" + formatdatetime("%y%m%d",systime()) + ".csv"
   delay(1)
endwhile
```

4) Click **Apply**, then go to **Debug** - **Run this Sequence** from the DAQFactory main menu to start the sequence.

All the sequence does is every second reset the file name. The new file name has the date in it. 86399 times out of 86400 it basically does nothing, but at midnight, it will actually change the file to a new name. The logging set will see this and automatically close yesterday's file and start a new one with the new name.

Since the sequence is marked AutoStart, when your document is loaded into DAQFactory, the sequence will start automatically.

Sample file: ***LJGuideSamples\LogDaily.ctl***

5.4 Conditional logging and the export set

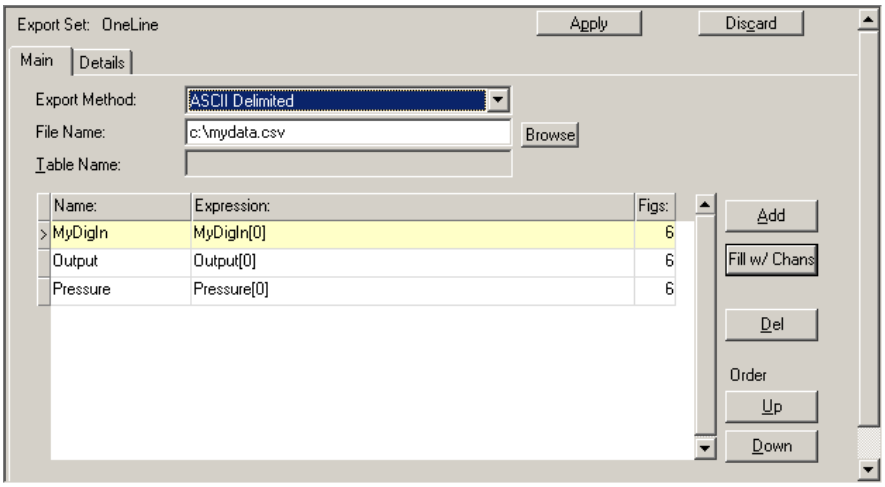
Logging sets work well with normal logging and batch logging since they typically run for longer periods of time. If, however, you need to log only when certain criteria are met, then you will need to use an Export Set instead. Continuing our examples, let us have the system log our channels whenever the Pressure goes above 90.

1) Right click on **EXPORT:** in the Workspace and select **Add Export Set**. Call it **`OneLine`**.

2) When the export set area appears, enter a **File Name**, such as **`C:\daqfactory\mydata.csv`**

3) Click on the **Fill w/ Chans** button.

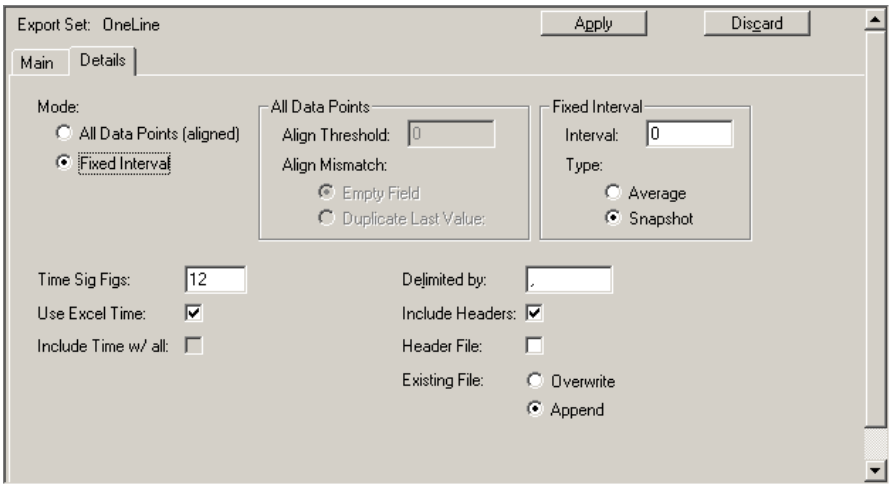
This button automatically fills the export set columns with each of your channels, setup to log the most recent reading only (notice the [0] after each expression in the table). This is the typical setup for conditional logging. Unlike logging sets, export sets can also log calculations and anything else you can describe in a DAQFactory Expression.



4) Click on the Details tab.

5) Select **Fixed Interval**, and then to the right under the **Fixed Interval** block, select **Snapshot**.

Fixed interval mode with Snapshot causes the export set to take a single snapshot of the current data irrespective of the time the data is acquired. This means that output channels, which are only updated in their history when they change values, will still appear on every logged data line. Since you only have one line, the Interval parameter does not apply.



6) Click **Apply**.

7) If not already expanded, click on the **+** next to **CHANNELS:** in the Workspace. Select the **Pressure** channel.

8) Click on the **Event** tab of the channel view. An event is script that is run whenever a new value arrives at the channel. Enter the following script:

```
if (Pressure[0] > 90)
   beginexport(OneLine)
endif
```

9) Click **Apply**.

That is all that is needed. The event code looks at the most recent reading of Pressure and if its over 90, it will start the export set. The export set will then log a single line of data.

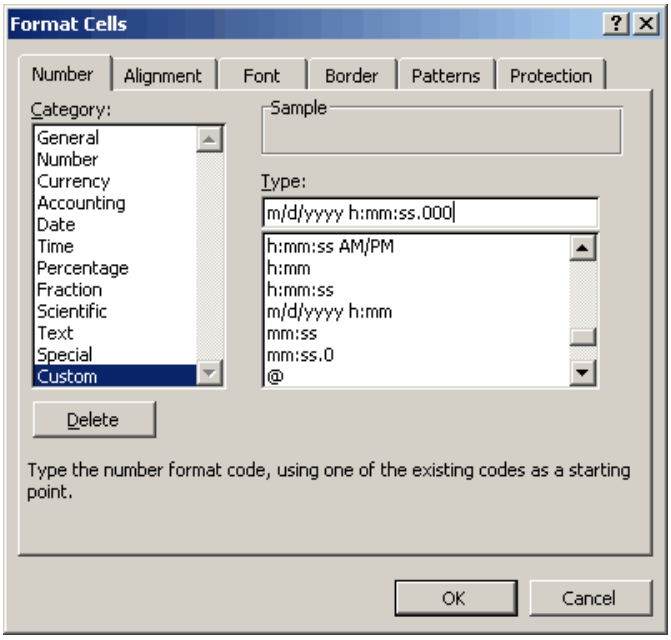
Sample file: ***LJGuideSamples\LogCondition.ctl***

5.5 Loading logged data into Excel

Assuming you didn't change the default delimiter, the logging files that we've created so far will all be in CSV or comma separated value format. This means that for a particular time, each channel value in the logging set will be listed, separated by commas, on a single line of text. The nice part about this is that Excel will directly load in a CSV file if you put a .csv extension on your file name. If you didn't, Excel will most likely display an import wizard. This is no worry, just make sure and select Comma as the delimiter instead of the default of Tab.

Once the file is loaded into Excel you will see all your data starting in the B column. The A column will always have the time. Assuming you didn't uncheck the **Use Excel Time?** option on the details page of the logging set, the A column will be numbers in the upper 30,000 range, or low 40,000 range if it is after July 2009. This is decimal days since 1900 and not terribly useful to us humans. To display the date and time in something we can read, right click on the A column header to select the entire column and select **Format Cells**. Then select any of the date / time formats available. Once this is done, you will probably have to resize the A column to see the complete date and time.

If you are doing reasonably fast data, the default date/time formats won't show values smaller than a second. To see millisecond values you will need to select the Custom format section, then select one of the date / time formats and add ".000" at the end:



6 Intro to scripting

6 Intro to scripting

6.1 Creating sequences

Much of what you can do in DAQFactory, such as acquiring data, logging it, and displaying it on the screen can be done without writing any script. However, to enjoy the full flexibility of DAQFactory, you will need to do at least a little basic scripting. You may think scripting is bad, but trust us: its the best way to get a computer to do exactly what you want. We could have made DAQFactory not have any scripting, but then every properties window for every feature would have thousands of choices, which would be much more cumbersome to use, and you still wouldn't have the flexibility of scripting. Fortunately, you can step into it slowly, using many of the built in features we've shown so far, but mixing in an occasional short script to do something unique.

We've seen some scripting already when we talked about conditional logging. This section will go over a few more of the basics of DAQFactory scripting so you'll understand the examples in the rest of this guide. It is not designed to be a complete overview of Sequences. For that you should read the DAQFactory User's Guide chapter on Sequences.

Script is used in a number of different places. In the conditional logging section we used script in Channel Events. In the batch logging section we had script in the Action for a screen component. The primary place for script, however, is in a sequence. A sequence is script that either executes standalone in the background, or as a function to another piece of script. Because sequences can run in the background while the rest of DAQFactory does its thing, you can create a separate sequence for each different function of your application and have them all run at the same time. For example, if you want a loop that switches a valve every 30 seconds, and another loop that sends an email every day, you would create two separate sequences and run them both at the same time.

Creating a new sequence is as simple as creating a new logging set. Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Enter a name for your sequence. Like every other name in DAQFactory, your sequence name must start with a letter and contain only letters, numbers and the underscore. Click **OK** and the new sequence is created and the sequence editor window displayed.

The sequence editor window is a simple text editor with some special features, like auto-indenting, that make it easier to write script. Before we jump into some of the script basics, a few important pointers. Our apologizes if we sound like we are preaching, but these are important points and will save you headaches down the road no matter whether you are scripting in DAQFactory, making webpages in PhP, or coding in Visual Basic. You'd be amazed at how many support calls we get because people didn't properly indent, can't figure out code they wrote a year ago, or don't see their decimal points:

Indenting:

Proper indenting of your script is very important. You should indent in 3 spaces on the line after any block level command (such as "if", "while", "for", "try", or "switch"), and indent out 3 spaces before the end of block commands (such as "endif", "endwhile", "endfor", or "endcatch"). Some lines do both, such as "else" and "catch". Here's a simple example of proper indenting. Don't worry about the script itself, just look at the indenting:

```
while(1)
   try
      if (x > 3)
         y = 1
      else
         y = 2
      endif
   catch()
      y = 4
   endcatch
   delay(1)
endwhile
```

Even if you don't understand any of the script above, you have to admit that this is easier to read than:

```
while(1)
try
if (x > 3)
y = 1
else
y = 2
endif
catch()
y = 4
endcatch
delay(1)
endwhile
```

OK, perhaps both are just unintelligible geek talk to you, but soon you will understand, and you should get into the habit of indenting properly from the start. There really are no excuses not to indent, unless, of course, your keyboard lacks a space bar.

Commenting:

We strongly recommend you put comments in your code. A comment is simply two slashes followed by whatever you'd like to say:

```
// this is a comment
global x   // x is an important variable because...
```

You may think that you'll remember how your script works a year from now, but without good comments, it is very unlikely. The DAQFactory source is probably one half code and one half comments.

Decimals:

Whenever you need to type a value that is less than 1 and greater than -1, you should always put a zero in front of the decimal point. In other words:

0.325 Correct

.325 Incorrect

-0.325 Correct

-.325 Incorrect

The reason for this is that when you have a page full of script the decimal place can easily be lost if there isn't that 0 in front. It may seem minor, but when its 2am and you can't figure out why your calculations are off because you confused 325 with .325 you'll wish you had gotten into this habit. Just look at this last sentence. If it was written properly as "confused with 325 with 0.325 you'll wish", it would be very obvious that the two numbers are not the same.

6.2 Scripting basics

6.2.1 Assignment

The most basic useful sequence is probably one that sets a few output channels to different values. Assignment is done by simply specifying the object to assign a value to, an equal sign, and the value to assign. For example:

```
MyOutputChannel = 3
AnotherOutput = 4 * 3 / 2
YetAnotherOutput = 5 / InputChannel[0]
```

The part to the left of the equal sign must be something that can take a value and cannot be an expression. This can typically be a channel or a variable. The part to the right of the equal sign can be any valid expression. If a sequence was created with just these three lines and run, the sequence would set MyOutputChannel to 3, AnotherOutput to 6 and YetAnotherOutput to 5 divided by the most recent value of InputChannel. The sequence would then stop.

6.2.2 Variables

Channels provide a place to store input and output readings and their associated histories. Often you'll need other places to store other values, such as calibration constants, counters, and flags. For this DAQFactory has variables. There are five types of variables in DAQFactory: global, local, private, registry, and static. Each variable type can be either a string or a number. DAQFactory also supports arrays of values. To use a variable you must create it first by declaring it. Declaring a variable is done in script by specifying the type of variable and the variable name. Like all other names in DAQFactory, a variable name must start with a letter and contain only letters, numbers and the underscore. For example, to declare a global variable called "Slope":

```
global slope
```

Or a private variable called "count":

```
private count
```

For string variables, we add the word string:

```
global string UserName
```

You can also do assignment while declaring the variable:

```
global slope = 3.92
```

In the examples that follow, will use a global variable called ID to store the LabJack ID, and then use this in place of a constant in all our script. Then, if we change the ID of the LabJack, we'll only have to change the script in one place.

So what do the data types mean?

global: these variables are accessible everywhere in DAQFactory, much like Channels.

private: these variables are only visible in the script that declared them. Use privates whenever possible, especially for counters used in for loops, etc.

local: these variables are used in custom devices, protocols, and user components. They are a bit more advanced and covered in the regular DAQFactory User's Guide

static: these variables are like privates, but unlike privates maintain their values when the sequence that declared them restarts. This is a rather advanced variable type.

registry: these variables are globals, but are actually stored in the Window's registry and so maintain their values when DAQFactory restarts. There are limitations. Numeric registry variables are signed 32 bit values so must be between -2147483647 and +2147483647. Registry variables also don't support arrays. Finally, registry variables are declared and accessed differently. Please see the DAQFactory User's Guide for more information.

6.2.3 Calling functions

There are a wide variety of functions available in DAQFactory scripting to allow you to do many advanced things like send email, FTP documents, popup windows, and of course perform basic math functions like Sin, Cos and Tan. Calling a function in DAQFactory is pretty much like every other scripting language, math tool, or even Excel. For example to get the Sin of 0.92 we would do:

```
sin(0.92)
```

This is a function that returns a value, in this case 0.7956016200364. There are also functions that perform an action, but don't return a value:

```
email.send()
```

Functions can take a varying number of parameters. The Sin() example has one parameter, 0.92, while the email. send() example has none. To specify more than one parameter in a function, you should separate each parameter by a comma:

```
smooth(mydata,10)
```

DAQFactory also allows you to create your own functions using Sequences. In fact, a sequence that you create in the Workspace can either be started on its own to run concurrently with other script, or called as a function from

other script. If it is started on its own, it cannot have any parameters and it will not return a value. When called as a function you can pass in up to 20 parameters and return a value. So, for example, if you decided you don't like the fact that sin() takes angles in radians and want to create a function called MySin() that takes degrees instead:

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**

2) Enter the name **MySin** and click **OK**.

3) In the script editor that appears, enter the following code:

```
function MySin(degs)
   private result = sin(degs * pi() / 180)
   return(result)
```

4) Click **Apply and Compile**, then click in the command part of the Command / Alert window (the bottom part) and put:

```
? MySin(90)
```

and hit Enter. You'll see it will display 1, the sin of 90 degrees.

A few points:

a) you don't have to do the function declaration as shown in the first line of the MySin script, but doing so allows you to name your parameters and thus makes your code cleaner.

b) we could easily have combined the 2nd and 3rd lines of the script and skipped creating a private variable, but we wanted to show variable declaration in action.

6.2.4 Conditional statements

Probably the most commonly used scripting statement is the if statement. The if statement allows you to check for a particular condition and perform a different action depending on whether that condition is true or not. For example, to set the Out channel to 3 if Pressure > 5 we would do:

```
if (Pressure[0] > 5)
   Out = 3
endif
```

We use [0] after Pressure because Pressure is a channel with history and we only want to look at the most recent reading of Pressu
We can also add an else statement to perform a different action if the condition is not true. So, if we want to set Out to 4 if Pressure isn't > 5, we would do:

```
if (Pressure[0] > 5)
   Out = 3
else
   Out = 4
endif
```

We can perform more than one statement inside an if too:

```
if (Pressure[0] > 5)
   Out = 3
   Valve = 1
endif
```

We can also nest if statements:

```
if (Pressure[0] > 5)
   Out = 3
   if (Temperature[0] < 80)
      Valve = 1
   else
      Valve = 0
   endif
else
   Out = 4
endif
```

This last example will set Out to 3 if Pressure > 5, and to 4 if Pressure <= 5. If, and only if, Pressure > 5, it will also set Valve to 1 if Temperature < 80, and to 0 if Temperature >= 80. If Pressure <= 5, Valve will not be changed. You have to admit its easier to understand in script than in the last 3 sentences! (but only because the script is properly indented...)

6.2.5 Loops and Delay

Another common scripting element is the loop. There are several ways to do a loop in DAQFactory, but the most common is probably the while statement, especially while(1). The while / endwhile block will execute a group of statements as long as the while statement is true. So:

```
while (Pressure[0] > 5)
   ? "Pressure High!!!!"
   delay(1)
endwhile
```

This loop will, as long as Pressure > 5, print out a statement every second. Once Pressure goes <= 5, the code after the endwhile (if any) will execute.

The most common while loop is while(1). Since 1 is the same as true, this loop will execute forever, or at least until the sequence that contains it is stopped from outside. Because it executes forever, you should really only use while (1) inside of a sequence and not inside of a component action or an event. That said, its great for doing things like calibration loops:

```
while (1)
   CalValve = 0
   delay(600)
   CalValve = 1
   delay(60)
endwhile
```

The above loop will set the CalValve off for 10 minutes, then on for 1 minute and repeat this as long as the sequence runs.

Its important to notice that we pretty much always put a delay() inside of our loops. Delay() will pause the execution of the script for the specified number of seconds. Without the delay() the loop would run as fast as your computer will allow and probably starve out Windows, making it look like DAQFactory has hung.

7 Some Common Tasks

7 Some Common Tasks

7.1 Doing things based on an input

Often one will want to monitor an input and then perform an action depending on what that input reading is. For example, we could be monitoring the temperature of a greenhouse and want to turn a heater on when it gets cold, and off when it gets hot. This is a basic thermostat. Assuming we had two channels, "Temperature", an analog input which has been converted to degrees C, and "HeaterPower" which is a digital output, to create a simple thermostat that turns on the heater at 15C and off at 22C we'd do this:

1) Click on the **+** next to **CHANNELS:** in the Workspace to expand this item and show all the channels.

2) Click on the **Temperature** channel.

3) Select the **Event** tab. Here we can enter script that executes every time a new reading occurs on the Temperature channel. Since this script can delay the readings, we need to make sure its short and fast.

4) Enter the following script:

```
if ((Temperature[0] < 15) && (Temperature[1] >= 15))
   HeaterPower = 1
endif
if ((Temperature[0] > 22) && (Temperature[1] <= 22))
   HeaterPower = 0
endif
```

5) Click **Apply**. The thermostat will start working immediately, though it won't actually change the state of the heater until the temperature passes through one of the thresholds.

This example shows a couple things that are a least important to thermostat applications:

- Hysteresis: we could just have the heater go on if the temperature is below 15 and off it is above, but this usually does not work well. Your analog inputs will likely have at least a little noise, and so when the temperature is right around 15, the readings will tend to jump above and below the 15 threshold causing the heater to rapidly turn on and off. To avoid this, we use something called hysteresis. The heater comes on at 15 degrees, but doesn't shut off until 22. Unless your noise is 7 degrees, the heater will respond much better.
- Thresholding: another way to write this script would be:

  ```
  if (Temperature[0] < 15)
     HeaterPower = 1
  endif
  if (Temperature[0] > 22)
     HeaterPower = 0
  endif
  ```

  The difference is that in this second, shorter script, the heater power digital output is set every time Temperature is read and is above or below the given threshold. This may create quite a bit of overhead on the device. Even the UE9 takes a few milliseconds to perform each action and if you are doing a lot of different things with the device, this can add up.

  So, in the original script, we look at the most recent point, [0], and the next most recent point, [1], and only adjust the HeaterPower output if they are on opposite sides of the threshold, meaning the temperature has just dropped below, or risen above the threshold. Once this occurs, the HeaterPower output won't be reset until the other threshold is crossed.

Sample file: ***LJGuideSamples\Thermostat.ctl***

7.2 Sending email out of DAQFactory

Email requires DAQFactory Starter or higher and will not work in DAQFactory Express.

Sending email out of DAQFactory is quite simple. You have to set a couple system variables and then call Email.

Send(). First, however, you need to know your SMTP server's address, username and password. This can usually be pulled out of your normal email program. Before we get into it, though, please note that DAQFactory does not support SSL, so won't work with gmail accounts since they require an SSL connection.

Since most of the email settings are the same for every email, we recommend putting that code in a sequence marked Auto-Start instead of calling it every time. It doesn't really matter, but it makes things more efficient. So:

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Give it the name **StartUp** and click **OK**.

2) In the sequence editor, check the box labeled **Auto-Start**, then put the following in the script, adjusting for your server settings:

```
email.strHost = "mail.myISP.com"
email.strUserName = "bob@myISP.com"
email.strPassword = "password"
email.strReplyAddress = "bob@myISP.com"
email.strAuthenticate = "AuthLogin"
```

You may need to tweak this last item. Depending on your server, it could be "NoLogin", "AuthLogin" or "LoginPlain". Also, most servers require a ReplyAddress, so make sure and set this one as well.

3) Click **Apply**, then go to **Debug - Run this Sequence** from the DAQFactory menu to actually set these settings.

Now that the basics are set, you can create code to actually send the email. Unless you are only going to send a message from one place, you may want to create a little Sequence function you can call to send an email with a given message. For example, if you are sending messages for alarms:

4) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Give it the name **SendEmail** and click **OK**.

Note we can't use "Email" as a name because it is used by DAQFactory.

5) In the sequence editor, put the following script, adjusting as needed:

```
function SendEmail(string Message)
   email.strTo = "my@myhome.com"
   email.strCC = "theboss@hishome.com"
   email.strSubject = "ALARM!!!!!"
   email.strBody = message
   email.Send()
```

Now that you have that function you can call it from elsewhere in DAQFactory. For example, lets say we want to send an email whenever our Pressure channel goes above 80 (from our earlier examples):

6) Click on **Pressure** in the workspace under **CHANNELS:** (this assumes you still have the Pressure channel from a few chapters ago).

7) Select the **Event** tab and enter this script:

```
if ((Pressure[0] > 80) && (Pressure[1] <= 80))
   SendEmail("Pressure is greater than 80.  Its going to blow!!!!")
endif
```

8) Click **Apply**. Now if the pressure goes above 80, an email will be sent.

Now you may be wondering about the if() statement we used. If we had done: "if (Pressure[0] > 80)" instead, then DAQFactory would fire off an email for every reading of pressure that is over 80. Instead, we are only sending an email as the pressure goes over 80. Pressure[0] is the most recent reading, and Pressure[1] is the next most recent reading. && means "AND", so only if the most recent reading is > 80, AND the next most recent reading is <= 80 will an email be generated.

Note: email takes a finite amount of time and is done asynchronously, meaning when you do email.send() to send the email, the email is sent in the background and the function returns immediately. Because of this, you should be careful not to consecutively call email.send() too quickly. Although it depends on your server, at least 5 seconds between sends is recommended.

Attaching files:

DAQFactory email also supports attaching of files. This is great if you want to be able to send the daily logs every night. We talked about how to create daily files in the Logging chapter. To attach the daily log to an email requires us to expand the script we made to change the logging file. Here's the modified script, with the assumption that you've set the To, Subject and Body elsewhere:

```
logging.mylog.strFileName = "c:\mydata_" + formatdatetime("%y%m%d",systime()) + ".csv"
private string lastdate = formatdatetime("%d",systime())
while(1)
   if (formatdatetime("%d",systime() != lastdate))
      // save the attachment file BEFORE we change the logging set name
      email.strFile = logging.mylog.strFileName
      logging.mylog.strFileName = "c:\mydata_" + formatdatetime("%y%m%d",systime()) + ".csv"
      // give the logging set time to close yesterday's file before we send the email
      delay(10)
      email.send()
   endif
   delay(1)
endwhile
```

7.3 Uploading data using FTP

FTP requires DAQFactory Starter or higher and will not work in DAQFactory Express.

Uploading data using DAQFactory FTP is very similar to Email. The variables are slightly different and all start with FTP, but the general concept is the same. You'll obviously need to know your FTP server address, user name and password. If your goal is to view your data from a web browser, you might consider the DAQConnect service described in section 7.5.

To demonstrate FTP, we'll use the email attachment example we did at the end of the last section and change it to do FTP instead, including all the initialization code too:

```
ftp.strServer = "ftp.myServer.com"
ftp.strUserName = "ftpUser"
ftp.strPassword = "password"
logging.mylog.strFileName = "c:\mydata_" + formatdatetime("%y%m%d",systime()) + ".csv"
private string lastdate = formatdatetime("%d",systime())
while(1)
   if (formatdatetime("%d",systime() != lastdate))
      // save the attachment file BEFORE we change the logging set name
      ftp.strLocalFile = logging.mylog.strFileName
      ftp.strRemoteFile = logging.mylog.strFileName  // you may need to change this if your server doesn't like
      logging.mylog.strFileName = "c:\mydata_" + formatdatetime("%y%m%d",systime()) + ".csv"
      // give the logging set time to close yesterday's file before we send the email
      delay(10)
      ftp.Upload()
   endif
   delay(1)
endwhile
```

7.4 Performing a ramped output

We've talked a little about setting an output from a screen control in section 4.6, but this requires user input. We also talked about setting an output based on an input in section 7.1. In this section we learned that you can easily set an output channel by simply assigning a value to it. So, to set a DAC channel named ValvePosition to 3 volts (assuming no Conversion exists on the channel), we would simply do:

ValvePosition = 3

You can also apply conversions to outputs like we did with inputs in section 4.3, however, the conversions work in reverse. For inputs, the conversion takes the raw voltage (counts or other units) from the LabJack and converts it into engineering units like temperature or pressure. For outputs, the conversion takes the engineering units and converts into voltage. So, for if we had a proportional valve that takes a 0 to 5V signal and we want to be able to

specify the percentage open, where 0% = 0V and 100% = 5V, the conversion would be:

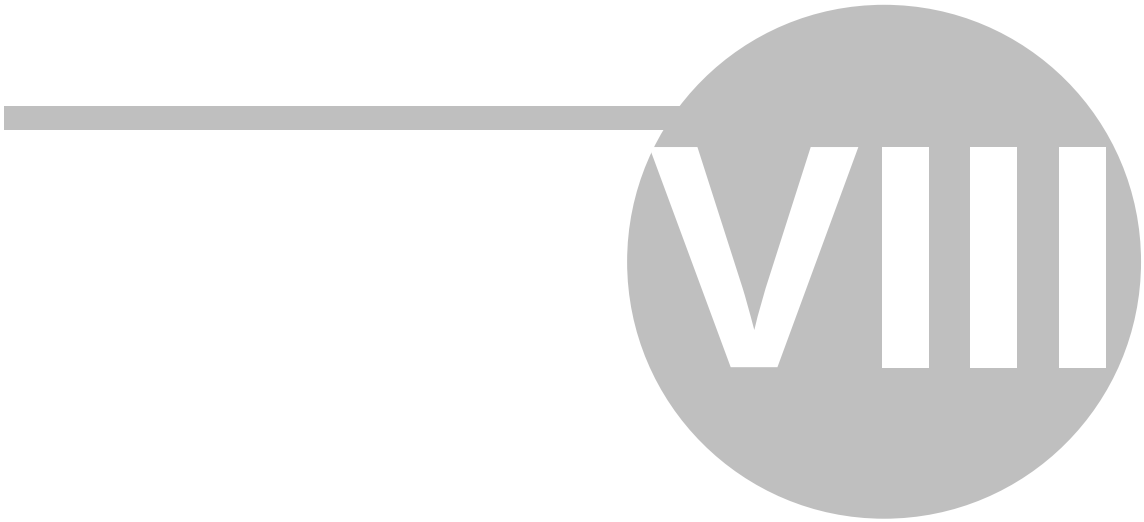
Value / 20

Then, to open the valve to 60% we could just do:

ValvePosition = 60

and 3 volts would be outputted from the DAC of the LabJack.

8 Calling the LabJackUD



8 Calling the LabJackUD

8.1 Using() and include() for LabJack

There are two important commands to make communicating with a LabJack using script easier. With these commands, script in DAQFactory for communicating with a LabJack will look almost identical to the psuedo-code that is in the LabJack User's Guide, and any C examples provided by LabJack. We recommend putting these two functions at the top of a sequence setup as AutoStart, so that it runs when your document is loaded. All subsequent samples assume this has been done. To do this:

1) Right click on **SEQUENCES:** in the workspace and select **Add Sequence**

2) Type in a name, say **StartUp** for the new sequence and click **OK**

3) The sequence editor will appear. Click in it and type the following two lines:

```
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
```

You may need to change the path in the second line to match the location of your labjackud.h file if you did not install the LabJack drivers in their default location.

4) Check the box just above the editor that says **Auto-Start**, then click **Apply** to save your changes.

5) If you are going to continue to work on the document without reloading it, you'll need to manually run this sequence, so right click on the sequence name, **StartUp**, in the Workspace and select **Begin Sequence**. The sequence will run instantly, so you won't really be able to tell that it ran.

Next time you load this document, the sequence will run and you are ready to go.

Now an explanation:

using("device.labjack.")

DAQFactory supports a wide variety of devices. Each device has different functions available to it. In LabJack's case, some of the functions are AddRequest, GoOne, etc. Because it is possible for two different devices to use the same function names, you normally have to prefix all your LabJack function calls with device.labjack. For example:

```
device.labjack.GoOne(1)
```

If you are only using a LabJack, this extra typing can be a pain, and makes code harder to read, so we allow you to specify different objects that don't require the object designation. Technically speaking this is bringing the function into the global namespace, but you don't have to worry about that. Just know that once you do the command in a document:

```
using("device.labjack.")
```

You don't have to put device.labjack. in front of the LabJack functions.

This resets when you load a new document, so you'll need to put this function call in an AutoStart sequence for each of your documents.

include("c:\program files\labjack\drivers\labjackud.h")

The LabJack Universal Driver uses a large number of constants to allow you access to the rather large number of features in these devices. To get access to these constants, we could hard code them into DAQFactory, but the folks at LabJack are constantly improving their devices and adding more features. With more features comes more constants and we simply can't keep up. So, with this include() call you are loading all the constants from the LabJack supplied file. That way, you always get the latest LabJack capabilities, without having to upgrade DAQFactory. This function works just like a C include, except that it only pulls out numeric constants.

Once the command has run, you can go to the Command / Alert window and in the bottom command area type:

```
? LJ_ioADD_STREAM_CHANNEL
```

hit Enter, and it will display 200, the value of this constant.

It should be noted that while C is a case sensitive language, DAQFactory is not, so **`? lj_ioadd_stream_channel`** will work just as well. That said, it will probably make your code more readable if you keep the case used in the constants.

8.2 Doing configuration steps

As we showed in the chapter Basic I/O, there is a lot you can do with DAQFactory and your LabJack without having to do any scripting. More than likely you will move into a little scripting when you want to tweak the configuration of your LabJack. This is often done in a sequence marked Auto-Start so the configuration is done automatically when your DAQFactory document loads. The format of these configuration commands are almost identical to the code shown in the LabJack User's Manual, as long as you use the using() and include() functions we described in the last section. So if we wanted to configure the UE9 analog inputs for 14 bit and channels 2 and 3 to bipolar 5V range:

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Give it the name **`StartUp`**

2) The sequence editor window will appear. Check the **AutoStart** at the top, and then in the window type in the following script:

```
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
AddRequest(0, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0)
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 14, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0)
GoOne(0)
```

3) Click on **Apply and Compile** to save your changes.

> When you specify AutoStart, the sequence is run when the document is loaded. It is not rerun every time you save changes, so you'll need to run the sequence at this point.

4) Right click on the sequence name **StartUp** in the workspace and select **Begin Sequence**. This will run the sequence and configure the LabJack. If you change this sequence, for example, to set the resolution to 16 bit, you will need to rerun the sequence to actually send the commands to the LabJack.

A few comments:

- This code uses ID 0, which means First Found in DAQFactory. If you have multiple LabJacks, you'll need to replace the 0 in the AddRequest() and GoOne() functions with your ID. Unlike the code in the LabJack User's Manual, you do not need to perform an Open() on the LabJack as DAQFactory will automatically open and find the desired LabJack as long as its connected via USB. If connected over Ethernet, please see the section on Ethernet setup.
- You may need to change the path in the second line to match the location of your labjackud.h file if you did not install the LabJack drivers in their default location.
- The LJ_ioPIN_CONFIGURATION_RESET ensures that the LabJack is in its original state before we configure it. This is a good precaution and should be included after your include() line even if you aren't going to do any other configuration in your AutoStart sequence. You can use ePut instead in this case since you are only doing one command:

```
ePut(0, LJ_ioPIN_CONFIGURATION_RESET, 0, 0, 0)
```

- You could also use the ePut() function, but its generally better to use AddRequest() and GoOne() when you need to perform more than one command at a time.
- Please review the LabJack User's Manual and the LabJackUD.h file for information on all the constants and configuration commands possible. There are also examples in the appropriate sections later in this document.

8.3 Error handling with OnAlert

There are several ways to handle LabJack errors from within DAQFactory. The correct way depends a bit on what you are doing and whether you actually want to do something if an error occurs.

If you are just performing basic I/O using Channels without any scripting, then any LabJack errors will appear in the Command / Alert window. Streaming errors also appear this way. If you cannot see the Command / Alert window, go to **View** - **Command / Alert** from the DAQFactory main menu. This allows you to see the errors, but you can't perform any automated action if an error occurs.

If you need to monitor errors automatically and perform an action when one occurs, then you should use the **OnAlert** DAQFactory event. As an example, here is how you would make the background of Page_0 turn red if a LabJack error occurs, resetting to white with a button press acknowledgement:

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Call the new sequence **`OnAlert`**

2) In the sequence editor that appears, enter the following script:

```
if (find(strAlert,"D0050:",0) != -1)
   page.page_0.backcolor = rgb(255,0,0)
endif
```

3) Click on **Apply and Compile** to save your changes

Now, to acknowledge the error and make the screen white again, we need a button:

4) Click on **Page_0** in the Workspace under **PAGES:**

5) Right click on the page somewhere and select **Buttons & Switches** and then **Button**

6) Right click on the new button and select Properties.... Type **`Acknowledge`** for the **Caption**

7) Click on the **Action** tab, then select **Quick Sequence** for the action.

8) In the sequence editor under Quick Sequence enter:

```
page.page_1.backcolor = rgb(255,255,255)
```

9) Click **OK** to save your changes.

Now if you have a few channels reading from the LabJack and an error occurs, Page_0 will turn red until the error stops and you click on the Acknowledge button you created. The easiest way to see this is to simply unplug the LabJack, then plug it back in.

*Sample file: **LJGuideSamples\OnAlert.ctl***

How it works:

If you create a sequence called OnAlert, it will be called whenever a new alert is generated by DAQFactory and passes in strAlert as a private variable, which is the actual error message. Alerts almost always start with a letter and 4 digit code. For device alerts, the letter is always "D" and the 4 digit code is unique to the particular device. In the case of the U3 / UE9 driver, it is always 0050. The rest of the message can vary, but for the LabJack, you will get an error code afterwards which can be used to further parse for a particular code. The format of LabJack errors is:

```
D0050:yy:xxxx: Message
```

where xxxx is a 4 digit error code, and y is the device ID / number that caused the error, or 99 if its a generic, non-device specific error.

The exception to this are errors that occur from Channel Timing. These errors have the timing information in front.

So, in our OnAlert sequence, we look for the string "D0050:" and if it is found, then we know its a LabJack error. Find() returns -1 if not found and a positive number otherwise. Then we simply set the background color of Page_0 to red using the RGB() function. The RGB() function takes three parameters, the amount of red, the amount of green and amount of blue, each from 0 - 255, and returns the full color value.

When using Channels to read values from your LabJack, DAQFactory will continually retry the read at the Timing interval you specified in the channel even if an error occurs. Because of this, you can end up with the same error

over and over again. This is why DAQFactory displays errors in the alert window and doesn't popup a separate window as many applications would. When you are doing error handling with OnAlert, you have to keep this in mind. You would not, for example, want to fire off an email with every error. If you were reading an input once a second and the LabJack was unplugged accidentally for 3 minutes, DAQFactory would send 180 emails, one for each failed read.

Since sending emails on error is a common task, here is probably the best way to solve the repetitive email dilemma: instead of firing off an email with every alert, add each new alert to the body and have a second sequence send out the email every so often if alerts have occurred. The details of sending email from DAQFactory are described in the next chapter, but to build up the email body you'd do this in the OnAlert:

```
if (find(strAlert,"D0050:",0) != -1)
   strEmailBody += strAlert + chr(13) + chr(10)
endif
```

The chr(13)+chr(10) add a carriage return / line feed to the end of each alert so that you don't get all your alerts strung out on a single line in your email.

Now you just need a sequence to fire off the email every so often, say 30 minutes. To do so, create a new sequence, probably marked AutoStart, with the following script:

```
global string strEmailBody = ""  // initialize the variable for accumulating errors
// put common email configuration stuff here:
email.strHost = "mail.mymail.com"
email.strUser = "me@mymail.com"
email.strPassword = "password"
email.strTo = "me@myhome.com
email.strFrom = "me@mymail.com"
// initial configuration done, so loop forever
while(1)
   // check if an alert has occurred
   if (strEmailBody != "")
      // if so, copy alerts into email body
      email.strBody = strEmailBody
      // reset variable for new alerts
      strEmailBody = ""
      // and send email
      email.Send()
   endif
   // wait 30 minutes and try again
   delay(1800)
endwhile
```

Note that our OnAlert script will probably generate its own errors until you run our email sending sequence. This is because the email sending sequence declares the strEmailBody variable, which until then doesn't exist and so we get errors in OnAlert.

8.4 Handling Disconnect / Reconnect

The most useful thing to catch in OnAlert is the pseudo-error codes LJE_DISCONNECT and LJE_RECONNECT. LJE_DISCONNECT is probably more useful as an alarm type alert to tell you that you accidentally unplugged the LabJack, and should maybe make the screen red or something similar. LJE_RECONNECT, however, provides a way for you to ensure that the LabJack is reset to a known state on power-up. Yes, the LabJack has the ability to set power-on defaults, but you may want different defaults depending on what document you are running. The best way to do this is to create a sequence will all the commands to set your default LabJack settings. You'll want to call this from a sequence marked AutoStart so the LabJack gets configured when you load the document. You'll then also want to add code like this to the OnAlert sequence / system event to call that configuration code when a reconnect occurs:

```
if (left(strAlert,13) == "D0050:00:2001") // 2001 is LJE_RECONNECT
   ConfigureLJ()
endif
```

This of course assumes that your configuration sequence is called ConfigureLJ. We can use Left() instead of Find() because Disconnect and Reconnect errors never come from Channel Timing loops.

Now, if you have multiple LabJacks, you'll also need to look at which LabJack got reconnected. In the example above, we assumed a D# / ID of 0, meaning first found. To further parse it:

```
if (left(strAlert,6) == "D0050:") // we have a labjack error
   private ID = strtodouble(mid(strAlert,6,2)) // retrieve ID
   if (mid(strAlert,9,4) == "2001") // reconnect
      ConfigureLJ(ID)
   endif
endif
```

This assumes that ConfigureLJ is setup to take the desired ID as the parameter. You might instead want to use a separate sequence for each ID.

This code just uses basic string manipulation functions to pull out the appropriate information:

left(s, n) returns the first **n** characters from the string **s**.

mid(s, i, n) returns **n** characters from character number **i** in the string **s**. Like everything else in DAQFactory, i is zero indexed, so the first character of the string is i = 0.

strtodouble(s) converts the given string s into a number.

8.5 Error handling in script

OnAlert works great for errors triggered by Channel Timing, by streaming, and to handle disconnects. If, however, you are using the various LabJack functions like AddRequest(), GoOne(), eGet(), and others in script, then an alert won't appears in the Command / Alert window, and OnAlert won't be called. When working in script, then, you have two other choices. The first is to simply look at the return value for each call to AddRequest(), GoOne(), GetResult (), etc. If an error occurred in calling the function, the error code is returned. Since 0 is LJE_NOERROR, which also means "false" in DAQFactory, we can simply look for a non-zero (or "true" in DAQFactory) value:

```
private err
err = AddRequest(...)
if (err)
   // error occurred!
endif
```

This method is OK for single functions like eGet and ePut, but can get real cumbersome if you have a lot of function calls. Fortunately, you can use the GetNextError() function instead. This function allows you to cycle through any errors in a block of LabJack function calls. A block is a group of AddRequest() followed by a single GoOne() and optionally any GetResult() calls. With each successive call to GetNextError(), the next error from the list of errors that occurred in the block is returned along with some details. Once the function returns LJE_NOERROR (which is just 0), then we know that there are no more errors. Here's an example:

```
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 14, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0)
GoOne(0)
private err
private ch
private io
while(GetNextError(0,@io,@ch,@err))
   ? Format("Error occurred on channel type: %d, io type: %d, code: %d",ch,io,err)
endwhile
```

The first three lines are the calls to the LabJack driver to configure the first found LabJack. We then declare three private variables to hold our error information. The GetNextError() function shows our first example of passing variables by reference in DAQFactory, which is used in many of the LabJack functions. Putting the @ in front of the parameter when calling GetNextError() indicates that the reference to that variable is being passed to the function, allowing the function to actually change the value of that variable. GetNextError() is a little unique in that it both returns the error code and sets a variable to the error code in a single call. This allows us to look for LJE_NOERROR in our loop, while still retrieving the error code.

```
while(GetNextError(0,@ch,@io,@err))
```

This line loops for as long as GetNextError() returns a non-zero value, meaning there are errors. If no errors occurred, the code inside the loop will not execute. If an error did occur, the I/O type (for example,

LJ_ioPUT_AIN_RANGE), the channel (for example 2), and the error code are put in our private variables. The next line:

```
   ? Format("Error occurred on channel type: %d, io type: %d, code: %d",ch,io,err)
```

Simply prints a formatted message with those values. Please see the DAQFactory User's Guide on using the Format () function. You may want to perform something more sophisticated when an error occurs.

If you want to the same error handling for all your script, you can put the error handling code in a separate function. To do so, create a new sequence and put the error handling code there:

```
private err
private ch
private io
while(GetNextError(0,@io,@ch,@err))
   ? Format("Error occurred on channel type: %d, io type: %d, code: %d",ch,io,err)
endwhile
```

Then, when you want to check for errors, simply call the sequence as a function. For example, if we called our new sequence "ErrorHandler", our original script would become:

```
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 14, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0)
GoOne(0)
ErrorHandler()
```

Note: A single call to eGet() or ePut() or any of the other e functions is a block, so you should stick with looking at the return value of these functions to determine if an error occurred.

This type of error handling is demonstrated in the example files for all the LabJack specific examples in the following sections.

9 Analog and Digital I/O

9 Analog and Digital I/O

9.1 Low speed acquisition < 100hz

9.1.1 Introduction

Because Windows needs to occasionally use the CPU in your computer to redraw the screen, process the mouse, and perform other tasks, you generally cannot read inputs or control outputs faster than 100hz using the computer as the timer. Of course with the new multicore CPU's, you may be able to tweak extra speed out DAQFactory's polling loops, essentially putting the acquisition on one core, and letting Windows use the other core for display. Doing this sort of thing requires more advanced techniques described in the last chapter. This section describes how to do software (i.e. DAQFactory) polled reading and writing of analog and digital inputs and outputs. The techniques are essentially the same for both analog and digital inputs and outputs.

9.1.2 The easy way - with channels

The easiest way to read analog and digital inputs at low speeds (i.e. < 100hz) or set analog and digital outputs is to use channels as we did in the Basic I/O chapter. No scripting is required, you can easily convert the readings using Conversions, and log it using Logging sets. For the U3, which allows the specification of the differential channel, you can simply put the channel number for the negative side in the Quick Note / Special / OPC column of the channel. If you need more advanced configuration, you can still use channels, combined with some basic scripting to send the configuration commands to the LabJack. For example, to set the UE9 resolution to 14 bit and the range on channels 2 and 3 to +/-5V:

```
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 14, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 2, LJ_rgBIP5V, 0, 0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 3, LJ_rgBIP5V, 0, 0)
GoOne(0)
```

This was explained earlier in the chapter on scripting.

9.1.3 Basic scripting using eGet

Channels, however, aren't always the best choice, and sometimes you need script. The next step up from channels is to use basic scripting and the eGet() function to retrieve a single value, or ePut() to set a single value. For example, to read channel 1 and then put the result into a channel called MyChannel we would do:

```
private err
private val
private string message
while(1)
   err = eGet(0, LJ_ioGET_AIN, 1, @val, 0)
   if (err)
      ErrorToString(err, @message)
      ? message
   else
      MyChannel.AddValue(val)
   endif
   delay(1)
endwhile
```

Now truthfully, this next code snippet does the exact same thing, provided MyChannel is setup to read channel 1:

```
while(1)
    read(MyChannel)
    delay(1)
endwhile
```

The read() function allows you to trigger the reading of a channel from script instead of using the Channel's Timing parameter. But, this code does not allow for any direct error handling, and of course doesn't demonstrate the eGet function! eGet is also more useful when you don't know your channel numbers at runtime. In the first example, we used scalar values in our eGet() function call, but there is no reason why you couldn't use variables that could then be changed from elsewhere:

```
err = eGet(ID, LJ_ioGET_AIN, chan, @val, 0)
```

Another thing the first example shows is the AddValue() function of the channel MyChannel. This function essentially stuffs a value into a channel. This allows you to utilize the benefits of a channel (history, easy logging, etc), without actually using the channel Timing to trigger the reads. In this case, we are putting the result of the eGet call into MyChannel. MyChannel does not have to have the same channel number, I/O type, or even be Device Type "LabJack".

Note: if using the U3, you will need to configure the pins as analog inputs. With Channels, this is done automatically for you, but when scripting you have to do it yourself. It only takes one line to enable a channel:

```
ePut(ID, LJ_ioPUT_ANALOG_ENABLE_BIT, 1, 1, 0)
```

Sample file: ***LJGuideSamples\eGet.ctl***

9.1.4 More advanced using Add / Go / Get

eGet can be handy for very simple scripts, but when doing a number of different requests, it is much better to use the AddRequest() / GoOne() / GetResult() functions. So, if we wanted to read channels 1 through 3 we could do:

```
eGet(0, LJ_ioGET_AIN, 1, @val1, 0)
eGet(0, LJ_ioGET_AIN, 2, @val2, 0)
eGet(0, LJ_ioGET_AIN, 3, @val3, 0)
```

or we could do:

```
AddRequest(0, LJ_ioGET_AIN, 1, 0, 0, 0)
AddRequest(0, LJ_ioGET_AIN, 2, 0, 0, 0)
AddRequest(0, LJ_ioGET_AIN, 3, 0, 0, 0)
GoOne(0)
GetResult(0, LJ_ioGET_AIN, 1, @val1)
GetResult(0, LJ_ioGET_AIN, 2, @val2)
GetResult(0, LJ_ioGET_AIN, 3, @val3)
```

OK, you may think that eGet looks much easier since its only three lines of code vs. seven, and it is easier on a basic level, but with ease you lose flexibility and efficiency. Using the second method is more efficient internally. The second method also allows you to do error handling easier using the GetNextError() function. With error handling, the eGet code ends up looking like this:

```
err = eGet(0, LJ_ioGET_AIN, 1, @val1, 0)
if (err)
   ... error!
endif
err = eGet(0, LJ_ioGET_AIN, 2, @val2, 0)
if (err)
   ... error!
endif
err = eGet(0, LJ_ioGET_AIN, 3, @val3, 0)
if (err)
   ... error!
endif
```

But the Add / Go / Get looks like this:

```
AddRequest(0, LJ_ioGET_AIN, 1, 0, 0, 0)
AddRequest(0, LJ_ioGET_AIN, 2, 0, 0, 0)
```

```
AddRequest(0, LJ_ioGET_AIN, 3, 0, 0, 0)
GoOne(0)
GetResult(0, LJ_ioGET_AIN, 1, @val1)
GetResult(0, LJ_ioGET_AIN, 2, @val2)
GetResult(0, LJ_ioGET_AIN, 3, @val3)

while (GetNextError(1,@io,@ch,@err)
   ... error!
endwhile
```

As you can see the second method is much cleaner and easier to read. The eGet() version would get worse and worse as you added more function calls. Using the second method also allows you to create a single error handler for the entire block, or as shown in the section on error handling, you can create a single function to do all your error handling for all your scripts.

Note: variable declarations are not shown in the above examples, but would be required. Likewise, ... error! would need to be replaced with script to actually do something in the case of an error.

9.1.5 Controlling outputs

We've talked a little about setting an output from a screen control in section 4.6, but this requires user input. We also talked about setting an output based on an input in section 7.1. In this section we saw that you can easily set an output channel by simply assigning a value to it. So, to set a DAC channel named ValvePosition to 3 volts (assuming no Conversion exists on the channel), we would simply do:

```
ValvePosition = 3
```

You can also apply conversions to outputs like we did with inputs in section 4.3, however, the conversions work in reverse. For inputs, the conversion takes the raw voltage (counts or other units) from the LabJack and converts it into engineering units like temperature or pressure. For outputs, the conversion takes the engineering units and converts into voltage. So, for if we had a proportional valve that takes a 0 to 5V signal and we want to be able to specify the percentage open, where 0% = 0V and 100% = 5V, the conversion would be:

```
Value / 20
```

Then, to open the valve to 60% we could just do:

```
ValvePosition = 60
```

and 3 volts would be outputted from the DAC of the LabJack.

Now as a further example, lets say we'd like to ramp the valve position from 0 to 100% over 60 seconds in steps of 1%. The script is quite simple:

```
// initialize valve position to 0
ValvePosition = 0
while (ValvePosition < 100)
   ValvePosition = ValvePosition + 1
   delay(0.6)
endwhile
```

Doing a ramp and soak is not much harder, just split out the while loops. Lets say we want to ramp to 40% in 20 seconds, soak for 10 seconds, ramp to 80% in 40 seconds, soak for 5 seconds, then ramp back down to 0 in 60 seconds:

```
// initialize valve position to 0
ValvePosition = 0
// ramp to 40
while (ValvePosition < 40)
   ValvePosition = ValvePosition + 1
   delay(0.5)
endwhile
delay(10) // soak
// ramp to 80
ValvePosition = 40 // this is to make the graph look good
```

```
while (ValvePosition < 80)
   ValvePosition++ // this is the same as VP = VP + 1
   delay(1)
endwhile
delay(5) // soak again
ValvePosition = 80
// ramp down to 0
while (ValvePosition > 0)
   ValvePosition--
   delay(0.75)
endwhile
```

One important note: these sequences won't work if you try and ramp past the highest value of the output. If you try and set an output to an invalid value you will get an Alert and the channel WILL NOT update, so the while() will never exit because ValvePosition would never reach the desired point.

Sample file: ***LJGuideSamples\RampSample.ctl***

9.2 High speed acquisition - streaming

9.2.1 Introduction

When you want to do things faster than 100hz, its usually best to let the LabJack perform the timing. The LabJack can then send data back in blocks and relieve your CPU from constantly having to do things. The LabJacks support streaming of both digital and analog inputs as well as timers and counters. This section explains how to setup and performed streamed data acquisition.

9.2.2 Basic streaming

When you want to read inputs at rates faster than 100hz, or at very precise intervals, it is usually best to use the LabJack's stream mode instead of having DAQFactory and the PC set the read rates. Anything above 100hz is difficult for a PC to perform since it has so many other tasks to do as well. To setup streaming in DAQFactory you will need to use a combination of Channels and sequence script. Streaming in DAQFactory is different from how the LabJack User Manual describes, as DAQFactory handles all the callback and data collection, putting the data into the appropriate channels. In this sample we'll stream 2 channels.

1) Start DAQFactory up with a new document.

2) Click on **CHANNELS:** in the Workspace to go to the channel table.

3) Add two new channels, **ChannelA** and **ChannelB**, both **Device Type** = **LabJack**, **D#** = **0**, **I/O Type** = **A to D**, and **Channel** numbers **2** and **3**. Set the **Timing** for both channels to **0**, and the **History:** set to **36000**.

3) Click on **Apply** to save your changes.

4) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Call the new sequence **StartStream**

5) In the sequence editor window that appears, enter the following script:

```
// standard initialization:
using("device.labjack")
include("c:\program files\LabJack\Drivers\labjackud.h")

// setup stream:
// set scan rate:
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chSTREAM_SCAN_FREQUENCY, 1000, 0, 0)
// setup channels to stream:
AddRequest(0, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 2, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 3, 0, 0, 0)
GoOne(0)

// start the stream:
global scanrate = 0
eGet(0,LJ_ioSTART_STREAM, 0, @scanrate, 0)
// scanrate now has the actual scanrate, which you can display on the screen if you want.
```

6) Click on **Apply and Compile** to save your script.

7) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Call the new sequence **StopStream** and enter the following script:

```
ePut(0,LJ_ioSTOP_STREAM, 0, 0, 0)
```

8) Click on **Apply and Compile** to save your script.

9) Click on **Page_0** under **PAGES:** in the workspace to display a blank page.

10) On that page, right click and select **Graphs- 2D Graph**.

11) While holding down the Ctrl key, click and drag the graph to move it to the top left corner of the page, then click and drag the bottom right corner of the graph to expand it to take up about 3/4 of the screen.

12) Right click on the graph and select **Properties…**. For the **Y Expression** put **ChannelA**. Click on **New Trace** and for the **Y Expression** put **ChannelB**. Click **OK** to close the properties window.

13) Right click somewhere on the empty part of the page and select **Buttons - Button**. Right click on the new button and select **Properties…**.

14) For **Caption**, put **Start**, then go to the **Action** tab and select **Start/Stop Sequence**, then select your **StartStream** sequence.

15) Repeat steps 13 and 14, but put **Stop** for the caption and **StopStream** for the sequence.

That is it. You should be able to click on the Start button and have streaming on channels 2 and 3 start up and be graphed. It is possible that the values will be off the scale of the graph, so you may want to click on the graph, then right click on the graph and select **AutoScale - AutoScale Y**.

One important point if you start tweaking this sample: the Channels that you created must have the same D# and channel number as the one you specified in the LJ_ioADD_STREAM_CHANNEL request. The I/O Type must be "A to D" as well, even if you are streaming digital inputs, timers or counters. If not then DAQFactory won't know where to put the data that is streaming in.

Note: make sure you configure your inputs before starting the stream. For the U3, this means you have to set the pins to analog input as shown in the example file.

Note: you should not change the LJ_chSTREAM_WAIT_MODE, as all waiting is handled internally. If you change this, you will most likely cause streaming to stop functioning.

Sample file: ***LJGuideSamples\BasicStream.ctl***

9.2.3 Streaming other inputs

You can stream other inputs besides the analog inputs of your LabJack. This is done by specifying special channel numbers when doing LJ_ioADD_STREAM_CHANNEL. The important part here is that even though the LabJack is actually streaming something other than an analog input, you ***MUST*** specify A to D for the I/O Type when creating your DAQFactory Channels to receive the data.

The available channel numbers are slightly different for each LabJack and listed here for your reference:

U3:

193 EIO_FIO
200 Timer0
201 Timer1
210 Counter0
211 Counter1
224 TC_Capture0
225 TC_Capture1
226 TC_Capture2
227 TC_Capture3

UE9:

193 EIO_FIO
194 MIO_CIO
200 Timer0
201 Timer1
202 Timer2
203 Timer3
204 Timer4
205 Timer5
210 Counter0
224 TC_Capture0
225 TC_Capture1
226 TC_Capture2
227 TC_Capture3
227 TC_Capture4
227 TC_Capture5
227 TC_Capture6

You may notice that in DAQFactory, we have multiple TC_Capture channel numbers, where the LabJack documentation only lists one. This is to allow you to stream the high order word of multiple timers and counters and keep the data separate in separate channels. Internally, they are exactly the same, so you have to specify the appropriate TC_Capture immediately following its Timer or Counter channel #. In other words, to read the entire 32 bits of Timers 0 and 1, you'd do:

```
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 200, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 224, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 201, 0, 0, 0)
AddRequest(0, LJ_ioADD_STREAM_CHANNEL, 225, 0, 0, 0)
```

9.2.4 Triggered

Triggered streaming is currently only supported by the UE9 and UE9-Pro.

Triggered streaming is similar to regular streaming except instead of using the internal LabJack clock to determine when a scan of the stream channels occurs, an external pulse triggers the scan. The interval between external pulses must be less than the maximum stream rate for the current input resolution. The external pulses do not need to occur at a constant interval. To enable external triggering, just add the following line of script before adding your stream channels using LJ_ioADD_STREAM_CHANNEL:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSTREAM_EXTERNAL_TRIGGER, 1, 0, 0)
```

The trigger input will be the first available FIO pin based on which timers and counters you have enabled.

The only problem with triggered streaming is that the time of each data point will be off. This is because the LabJack buffers the scans and DAQFactory doesn't actually get the data until a full packet occurs. DAQFactory doesn't realize this and assigns times based on an assumed interval. If you have the bandwidth, i.e. your pulses are slow enough that you aren't close to the stream interval limit, you can use the system timer mode of the timers to retrieve exact relative times of your scans. To do this, you need to setup a timer for system timer in, and then add the timer to the list of channels to stream. Depending on how long your experiment runs, you may be able to get away with only SYSTIMERLOW. For the UE9 at 750khz, the low timer will roll over every 5726 seconds. Here's how to do it:

1) Enable two Timers:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0)
```

2) Set the mode:

```
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmSYSTIMERLOW, 0, 0)
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 1, LJ_tmSYSTIMERHIGH, 0, 0)
GoOne(ID)
```

3) Set up the stream to stream analog input 2 and 3 in external trigger mode:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSTREAM_EXTERNAL_TRIGGER, 1, 0, 0)
// setup channels to stream:
AddRequest(ID, LJ_ioCLEAR_STREAM_CHANNELS, 0, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 3, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 4, 0, 0, 0)
```

4) Now we need to add our timers to the list of channels to stream. Make sure you use the order indicated:

```
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 200, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 224, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 201, 0, 0, 0)
AddRequest(ID, LJ_ioADD_STREAM_CHANNEL, 225, 0, 0, 0)
```

5) Now finish up the stream setup:

```
GoOne(ID)

// start the stream:
global scanrate = 0
eGet(ID,LJ_ioSTART_STREAM, 0, @scanrate, 0)
// scanrate now has the actual scanrate, which you can display on the screen if you want.
```

6) Create 4 channels to receive this timing data in addition to the 2 you created to receive the analog input. All 6 channels are **I/O type: A to D**, **Timing** = **0**. **Channel** #'s will be **3**, **4**, **200**, **201**, **224**, and **225**.

That completes it. When you run your script, the stream will start, streaming both analog inputs 2 and 3 as well as the system timer. A scan will occur every time a trigger is detected on FIO2. FIO0 and FIO1 are used by the 2 timers. With each scan, your six channels will update. The time associated with these channels will be incorrect, but channels 200, 224, 201, and 225 will contain the 4 words that make up the 64 bit system timer value. While this is not absolute time, it will give you relative time between each triggered scan. Just use the difference in counts divided by the system clock speed of 750khz for the UE9 to determine the actual number of seconds between scans. The best way to do this is to create a calculated V channel:

7) Right click on **CHANNELS:** under **V:** in the Workspace. Note this is note the same CHANNELS: that we've been clicking before. Select **Add V Channel**

8) Call the new channel **TheTime**

9) Click on the new channel in the workspace. In the **Expression** window, put:

```
(TimerLowLow + TimerLowHigh*2^16 + TimerHighLow * 2^32 + TimerHighHigh * 2^48)
```

This assumes you named channel 200 TimerLowLow, 224 TimerLowHigh, 201 TimerHighLow, and 225 TimerHighHigh. You also may want to put a divisor at the end to convert to seconds:

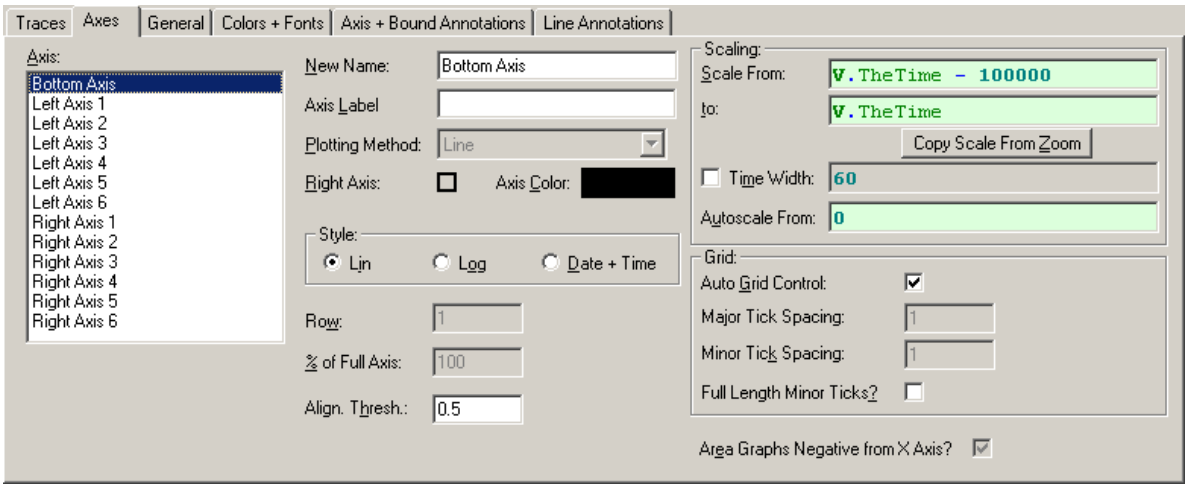
UE9: `(TimerLowLow + TimerLowHigh*2^16 + TimerHighLow * 2^32 + TimerHighHigh * 2^48) / 750000`

10) Click **Apply**.

At this point, you can reference this channel like you would any other, except putting V. in front of it. Instead of getting a channel reading, you'll get the result of the calculation. Since we didn't use any [0] notation, this is the entire array calculated from all the readings. If you want to graph your channels, you'd put:

```
V.TheTime
```

as the **X Expression** in place of Time. You'll need to change the bottom axis type to **Lin**, undo **Use Time Width**, and adjust the **Scale From** and **Scale To**:



All of this is shown, complete, in the sample file:

Sample file: ***LJGuideSamples\TriggeredStream.ctl***

9.2.5 Error handling for streaming

Streaming from a LabJack is what is called an asynchronous action. This means that the LabJack does its own thing and every so often it tells DAQFactory that there is new data or there is an error. For this reason, you cannot simply look at the error code returned by LJ_ioSTART_STREAM to catch all stream errors. This command may return a stream configuration type error, so you'll want to check for it using the same methods as the low speed acquisition, but will not handle errors in the actual stream. For this you have two choices:

1) You can create a simple sequence to retrieve the last stream error continuously and do something if it returns an error. The function **`GetLastStreamError()`** will return the code for the last stream error. This is reset when you start the stream. This, however, is not the best way to do this and will waste processor power.

2) You can use the DAQFactory OnAlert event, which only gets called when an actual error occurs. Using the event as an error handler is described in the previous section on error handling, and catching stream errors would be done the same way. One common error you might want to catch is if your LabJack accidentally gets unplugged while streaming. If you wanted to automatically restart streaming when it is reconnected you could do this in your OnAlert sequence:

```
if ((left(strAlert,10) == "D0050:00:2001") && (Streaming))  // 2001 is LJE_RECONNECT
   StartStream()
endif
```

Now, this assumes that you have a sequence called StartStream that will reconfigure the LabJack and actually restart the stream. It also assumes that you've created a global variable called "Streaming" that you set to 1 in StartStream, and to 0 in StopStream. This is so an accident unplug when you aren't streaming doesn't spontaneously start the streaming process. Finally this assumes you are using device number 0 / first found. Please see the section on OnAlert if you are using multiple LabJacks.

9.3 Thermocouples

Thermocouples are a very common way to measure temperature. They come in a wide variety of forms and have a large range. Do to their small voltage output and something called a cold junction, they are, however, slightly more challenging to read. Most applications need a temperature range of about -50 to +150C, though, and for these applications you might want to consider using a silicon type sensor, as they are cheaper, more accurate and much easier to use. Examples of these are the EI-1022 and EI-1034 sold by LabJack.

Thermocouples are simply two different metals that meet at a single point. Due to a general property of metals, a small voltage is generated that is proportional to the temperature of this junction. There are many different types of thermocouples, identified by a letter such as J type or K type which determine the metals used which then determines the temperature range and voltage output scaling. In all cases, this voltage is between about -5 and 5

millivolts, a rather small voltage.

It gets more complicated though: since the terminals on your DAQ device (the LabJack) are made of yet another metal, there are two more points where different metals are touching which are generating a voltage proportional to the temperature of this terminal. This is called the cold junction. In order to get an accurate thermocouple reading, you have to adjust for the cold junction. This is called cold junction compensation.

Finally, we should mention that most thermocouples are only accurate to about 1 or 2 degrees C, though calibration can help a little with this.

Now that you have the basics down we can talk about how to read a thermocouple. There are several choices depending on what LabJack you have:

U3: the U3 has a resolution of 12 bit and a range of +/-2.44, the minimum voltage you can read is about 1.2 millivolts (4.88 volts / 4096 steps). Because a thermocouple typically outputs around 40 microvolts per degC, you only get a precision of about 30 degrees! Obviously that isn't going to work so you have to use an amplifier such as the LJTIA or EI1040, both available from LabJack, to amplify that millivolt signal before it gets to the converter.

UE9: the UE9 has a resolution of 16 bit so has 65536 steps instead of the 4096 of the U3. This helps some. The UE9 also has some built in gain. Typically you get about 76 microvolts minimum step noise free, and 15 microvolts RMS. This means you get about 1 or 2 degrees C of resolution. This isn't bad and might work. For the exact specs, please review Appendix B of the UE9 Users Guide. If you need better than this, then you will need to get an LJTIA or EI1040 amplifier from LabJack to amplify the thermocouple signal. If you aren't in a rush for your measurements and the temperature is changing slowly, you can also use DAQFactory to oversample by taking multiple measurements and averaging them. This is done by checking the Avg? column of your input channel in the channel table and entering the number of oversample points in the #Avg column.

UE9 Pro: the UE9 Pro gets you an additional 2 bits of resolution over the UE9 (27 microvolts noise free minimum step, 5 microvolts RMS) which is enough to read a thermocouple directly with a reasonable amount of precision. You can use an amplifier as well if you want higher speed readings.

Now that we have that covered lets go over using both methods: unamplified and amplified.

9.3.1 Unamplified Thermocouple Readings

This applies only to the UE9 and UE9 Pro because the U3 doesn't have the precision to read the low voltages output by a thermocouple. Please see the last section (7.5) for a full explanation.

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Give it the name **StartUp** and click **OK**.

2) In the sequence editor, check the box labeled **Auto-Start**, then put the following in the script:

```
global offset = 0

using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")

AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 18, 0,0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 0, LJ_rgUNI5V, 0, 0)
GoOne(0)
```

The first line is used to adjust for any voltage offset in the signal. The next two lines are the standard lines required to access the LabJack UD from script. The next three lines set the gain and resolution of the UE9. We've chosen the most generic range for AIN0. Your UE9 might be able to use a tighter AIN range and achieve higher precision.

3) Click **Apply**, then go to **Debug - Run this Sequence** from the DAQFactory menu to actually set these settings.

4) Create a channel for AIN0 as described back in 3.1 to read the voltage.

5) To accurately read the thermocouple we also need to read the temperature of the cold junction which is essentially the temperature of the LabJack. This is internal A to D channel #133. Create another channel called CJC to read A to D channel #133. Click **Apply** to save your changes and start reading the inputs.

Now we can convert the thermocouple voltage input into temperature:

6) Click on **CONVERSIONS:** in the workspace, then **Add**. Call the conversion Thermocouple and put the following

Formula:

```
TypeK(Value - Offset, CJC[0]-273.15)
```

There are different formulas for each thermocouple type, such as TypeJ().

7) Go back to the channel table and for the AIN0 channel you created in step 4, select the Thermocouple conversion you just created and click **Apply**.

That is it. You can now display or plot the input channel. You can adjust the offset variable to adjust for any bias in the LabJack by changing it in the sequence we created in step 2 and rerunning it, or by creating a screen control to adjust its value (shown in the sample).

Sample file: ***LJGuideSamples\UE9_Thermocouple.ctl***

The sample doesn't use a conversion, but instead puts the formula in the components themselves. Conversions are typically easier since you typically only want to see the temperature, however, for calibration you often want the raw voltage and so can't use a conversion.

9.3.2 Amplified Thermocouple Readings

When using an amplifier, the steps are pretty much identical to the unamplified version except we need to adjust the input based on the amplification:

1) Right click on **SEQUENCES:** in the Workspace and select **Add Sequence**. Give it the name **StartUp** and click **OK**.

2) In the sequence editor, check the box labeled **Auto-Start**, then put the following in the script:

```
global offset = 0
global gain = 51
```

If using a UE9, you can add these lines as well:

```
using("device.labjack.")
include("c:\program files\labjack\drivers\labjackud.h")
AddRequest(0, LJ_ioPUT_CONFIG, LJ_chAIN_RESOLUTION, 18, 0,0)
AddRequest(0, LJ_ioPUT_AIN_RANGE, 0, LJ_rgUNI5V, 0, 0)
GoOne(0)
```

The first line is used to adjust for any voltage offset in the signal. The second to adjust for the gain of the amplifier. The next two optional UE9 lines are the standard lines required to access the LabJack UD from script. The next three lines set the gain and resolution of the UE9. We've chosen the most generic range for AIN0. Your UE9 might be able to use a tighter AIN range and achieve higher precision.

3) Click **Apply**, then go to **Debug - Run this Sequence** from the DAQFactory menu to actually set these settings.

4) Create a channel for AIN0 as described back in 3.1 to read the voltage.

5) To accurately read the thermocouple we also need to read the temperature of the cold junction which is essentially the temperature of the LabJack. This is internal A to D channel #133 for the UE9 and #30 for the U3. Create another channel called CJC to read A to D channel #133 if using a UE9 or #30 for a U3. Click **Apply** to save your changes and start reading the inputs.

Now we can convert the thermocouple voltage input into temperature:

6) Click on **CONVERSIONS:** in the workspace, then **Add**. Call the conversion Thermocouple and put the following Formula:

```
TypeK((Value - Offset)/gain, CJC[0]-273.15)
```

There are different formulas for each thermocouple type, such as TypeJ().

7) Go back to the channel table and for the AIN0 channel you created in step 4, select the Thermocouple conversion you just created and click **Apply**.

That is it. You can now display or plot the input channel. You can adjust the offset variable to adjust for any bias in the LabJack by changing it in the sequence we created in step 2 and rerunning it, or by creating a screen control to adjust its value (shown in the samples). Same thing with the gain.

It doesn't really matter what amplifier you use. You can just adjust the offset and gain appropriately.

Sample file: ***LJGuideSamples\U3_LJTIA_Thermocouple.ctl***

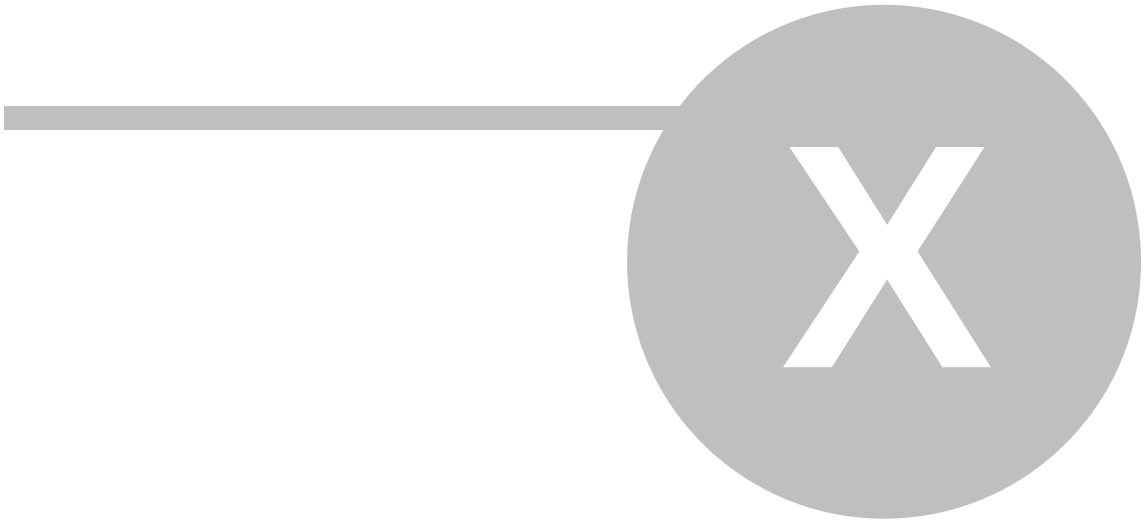
Sample file: ***LJGuideSamples\UE9_LJTIA_Thermocouple.ctl***

These sample doesn't use a conversion, but instead puts the formula in the components themselves. Conversions are typically easier since you typically only want to see the temperature, however, for calibration you often want the raw voltage and so can't use a conversion.

Sample file: ***LJGuideSamples\U3_LJTIA_Thermocouple_Conversion.ctl***

This is identical to the U3_LJTIA_Thermocouple sample, but uses a Conversion to convert from volts to temperature. You can see the Conversion by clicking CONVERSIONS: in the workspace. A Conversion converts the data as it comes in so the channel (FIO4) shows temperature immediately. You never see volts. Conversions can also be applied to multiple channels. That said, for thermocouples, you will probably want a unique conversion for each thermocouple since each input might have a slightly different Offset and Gain. In this case, you'd need to replace "Offset" and "Gain" in the conversion with the actual number for the corresponding input once you have figured these out.

10 Counters and Timers



10 Counters and Timers

10.1 Configuring

The counters and timers on the LabJack units are quite flexible. In order to configure them, you have to use some basic scripting, similar to the script we've seen so far. Because setting up a counter or timer typically requires setting multiple parameters, we recommend using the AddRequest() / GoOne() / GetResult() method instead of ePut (), but really its up to you. There are also several e functions you can use, namely eTCValues() and eTCConfig(), but these functions require you to define variables, so take as many, if not more steps than using AddRequest(). Internally, these functions call AddRequest() anyway, so we recommend just using AddRequest() from the start. eTCValues(), however, could be useful if you were using a number of timers and counters and not using Channels to store the results, but that is beyond the scope of this guide.

10.2 Reading values for counters and input timers

Reading the values of counters and timers can of course be done with script as well, using the LJ_ioGET_COUNTER and LJ_ioGET_TIMER commands. However, you can also use channels, which will perform the same command for you. If you create a new channel like you did for analog inputs, but select either the Timer or Counter I/O Type, DAQFactory will query the timer or counter value at the interval you specified. You still need to initialize and configure your timer or counter in script, but once configured you can use these two I/O types to perform the reads.

10.3 Basic Counter and Timer setup

To use the LabJack timers or counters you need to do some very basic setup.

Note: to enable this guide and the corresponding samples to work with all LabJacks, we use the System Clock (tcSYS) and a PIN_OFFSET of 4. Depending on your hardware, you should feel free to use other clocks and other pin offsets.

First, by default, once you enable a timer or counter, it will replace FIO0. If you'd prefer to keep FIO0 for analog or digital I/O, you can use the LJ_chTIMER_COUNTER_PIN_OFFSET, to select a different pin:

```
AddRequest (ID,  LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 1, 0, 0)
```

The above, after a GoOne(ID), will set put the first enabled timer or counter on FIO1 instead of FIO0. The general form of the command is:

```
LabJack ID, LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, FIO pin # for first timer/counter, 0, 0
```

Please note that as of hardware revision 1.3 of the U3, timers and counters will start at pin offset 4. Therefore a pin offset of 0 to 4 will all result in FIO4 being the first timer. A pin offset of 5 will result in FIO5 being the first timer. FIO0 through FIO3 will no longer be usable as timers or counters.

Counters:

To use counters, all you really need to do is enable the counter. This is done with LJ_ioPUT_COUNTER_ENABLE:

```
AddRequest(ID, LJ_ioPUT_COUNTER_ENABLE,0,1,0,0)
```

This will enable the first counter. To disable, do the same thing, but change the 1 to a 0. The general form of this command is:

```
Labjack ID, LJ_ioPUT_COUNTER_ENABLE, Counter #, Enable (1) or Disable (0), 0, 0
```

Once the counter is enabled, you can read the counter using a channel, putting the ID in for the D#, select an I/O type of Counter, and putting the counter number for the channel number. If you are looking for the number of counts in a certain time period, please make sure and read the next section on resetting the counter.

Sample file: ***LJGuideSamples\BasicCounter.ctl***

Timers:

Timers are slightly more complicated, mainly because they are a lot more flexible. There are a number of different timer modes and each has its own parameters and setup which is described in the following sections. A few common points though:

Like counters, you'll need to first enable the timers. The function to do so is very similar to counters except you are specifying how many timers to enable rather than enabling a specific timer. So, to enable two timers:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0)
```

After you have enabled the timers, you'll need to set which mode you'd like to use for each timer. For example:

```
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmPWM8, 0, 0)
```

will set the Timer mode of timer 0 to PWM8. The general form of this command is:

```
LabJack ID, LJ_ioPUT_TIMER_MODE, Timer #, Timer mode code, 0, 0
```

Possible Timer modes as of this writing include:

```
LJ_tmPWM16               // 16 bit PWM
LJ_tmPWM8                // 8 bit PWM
LJ_tmRISINGEDGES32       // 32-bit rising to rising edge measurement
LJ_tmFALLINGEDGES32      // 32-bit falling to falling edge measurement
LJ_tmDUTYCYCLE           // duty cycle measurement
LJ_tmFIRMCOUNTER         // firmware based rising edge counter
LJ_tmFIRMCOUNTERDEBOUNCE // firmware counter with debounce
LJ_tmFREQOUT             // frequency output
LJ_tmQUAD                // Quadrature
LJ_tmTIMERSTOP           // stops another timer after n pulses
LJ_tmSYSTIMERLOW         // read lower 32-bits of system timer
LJ_tmSYSTIMERHIGH        // read upper 32-bits of system timer
LJ_tmRISINGEDGES16       // 16-bit rising to rising edge measurement
LJ_tmFALLINGEDGES16      // 16-bit falling to falling edge measurement
```

Not all modes may be supported by all LabJacks. Please see the file LabJackUD.h in your LabJack installation directory for any new modes.

Finally, you'll probably need to set the clock base and divisor that the timer will use:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0)
```

These both follow the standard form of PUT_CONFIG:

```
LabJack ID, LJ_ioPUT_CONFIG, Parameter, Value, 0, 0
```

The clock divisor is an integer. For timer base, there are several constants defined:

```
LJ_tcSYS        // all: system clock, varies depending on device

LJ_tc750KHZ     // UE9: 750 khz

LJ_tc2MHZ       // U3: Hardware Version 1.20 or lower
LJ_tc6MHZ       // U3: Hardware Version 1.20 or lower
LJ_tc24MHZ      // U3: Hardware Version 1.20 or lower
LJ_tc500KHZ_DIV // U3: Hardware Version 1.20 or lower
LJ_tc2MHZ_DIV   // U3: Hardware Version 1.20 or lower
LJ_tc6MHZ_DIV   // U3: Hardware Version 1.20 or lower
LJ_tc24MHZ_DIV  // U3: Hardware Version 1.20 or lower

LJ_tc4MHZ       // U3: Hardware Version 1.21 or higher
LJ_tc12MHZ      // U3: Hardware Version 1.21 or higher
LJ_tc48MHZ      // U3: Hardware Version 1.21 or higher
LJ_tc1MHZ_DIV   // U3: Hardware Version 1.21 or higher
LJ_tc4MHZ_DIV   // U3: Hardware Version 1.21 or higher
LJ_tc12MHZ_DIV  // U3: Hardware Version 1.21 or higher
LJ_tc48MHZ_DIV  // U3: Hardware Version 1.21 or higher
```

Once again, not all clock bases are supported by all LabJacks and you should check the LabJackUD.h file for any new bases.

10.4 Resetting Counters

A common use for counters is to count the number of events that occur within a preset time, often within a second. In these cases, it is tempting to reset the counter after each interval. This, however, is not recommend because every time you reset the counter there is a very short dead period where no counts can be measured. At higher count rates you can easily miss counts. To avoid this, you should use the power of DAQFactory to calculate the difference between two consecutive readings to get a counts per interval reading rather than resetting the counter.

To do this, you will need two channels. First, we'll assume you have properly initialized your counter as described in the previous section. You also probably should reset the counter at startup using:

```
AddRequest(ID, LJ_ioPUT_COUNTER_RESET,0,0,0,0)
```

where the first 0 is the counter number to reset, in this case the first counter.

1) Create a channel to read the counter. Call it **RawCounts** or similar. This will be **Device Type = LabJack** of course, **D#** = LabJack ID, **I/O Type** of **Counter**, **Channel #** = desired counter or 0 on devices with only one counter. Set the **Timing** to whatever your desired interval is. For counts per second, put 1.00.

2) Create a second channel that will hold your counts per interval. Call it **Counts** for now. This channel will be **Device Type = Test**, **D#** = **0**, **I/O Type** = **A to D**, **Channel #** = **0**, and most importantly, **Timing** = **0**. Click **Apply** to save your new channels.

3) Click on the **+** next the **CHANNELS:** in the Workspace, then click on the **RawCounts** channel. When the channel view appears, click on the **Event** tab. Enter the following script:

```
Counts.AddValue(RawCounts[0] - RawCounts[1])
```

4) Click **Apply**. At this point, provided RawCounts is actually getting increasing counts, the Counts channel will have the interval counts. You can click on the **Table** tab to see this (after clicking on **Counts** in the Workspace).

The problem with the above method is that it doesn't account for counter roll over. On a 32 bit counter this happens at just over 4 billion counts, so before worrying about this, you might want to figure out how long it would take to accumulate that many counts and see if its worth worrying about. Remember that if you reset the counter at startup, you only have to worry about the amount of time DAQFactory is continuously running. If rollover does occur, all you will see is a single, negative interval counts measurement. You can post-calc the correct measurement by simply adding 4294967296 to this negative number. That is 2 raised to the 32 power.

But, if you don't want negative counts on rollover, you just have to change the Event from step 3 slightly:

```
if (RawCounts[0] > RawCounts[1])
   Counts.AddValue(RawCounts[0] - RawCounts[1])
else
   Counts.AddValue(RawCounts[0] - RawCounts[1] + 2^32)
endif
```

Of course if you have a 16 bit counter, you'll need to change the 32 to 16 so it adds 65536 instead.

Hertz measurements:

Finally, if instead of actual counts in an interval, you want a hertz measurement (counts per second), we just change the AddValue() lines to divide by the difference in time:

```
if (RawCounts[0] > RawCounts[1])
   Counts.AddValue((RawCounts[0] - RawCounts[1]) / (RawCounts.Time[0] - RawCounts.Time[1]))
else
   Counts.AddValue((RawCounts[0] - RawCounts[1] + 2^32) / (RawCounts.Time[0] - RawCounts.Time[1]))
endif
```

The nice part about this is that if DAQFactory gets delayed a few milliseconds before doing the read, the hertz measurement will be properly normalized.

Sample file: ***LJGuideSamples\BasicCounter.ctl***

10.5 Setting up specific timer modes

10.5.1 PWM out

There are two timer / counter modes for pulse width modulation (PWM), one is 16 bit, the other 8 bit. These, and the available timer clocks are described in your LabJack User's manual and vary depending on the device. The setup, however, is largely the same. Here's some sample script for setting up and starting a PWM8 on FIO4. As always, we assume you've done using() and include() someplace else and defined ID appropriately:

```
//Set the timer/counter pin offset to 4, which will put the first timer/counter on FIO4.
AddRequest (ID,  LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 4, 0, 0)

// use system clock so works on U3 and UE9:
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)

//Enable 1 timer.  It will use FIO0.
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 1, 0, 0)

//Configure Timer0 as 8-bit PWM.  Frequency will be 1M/256 = 3906 Hz.
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmPWM8, 0, 0)

//Set the PWM duty cycle to 50%.
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 0, 32768, 0, 0)

//Execute the requests.
GoOne(ID)
```

We've explained most of these commands already. The only one new is the LJ_ioPUT_TIMER_VALUE. This sets the PWM duty cycle. Even though we are using an 8 bit PWM, this takes a 16 bit number. 32768 is half way into a 16 bit unsigned integer, so this results in a 50% duty cycle PWM. The general form of this command is:

LabJack ID, LJ_ioPUT_TIMER_VALUE, Timer #, Value, 0, 0

The two modes are:

```
LJ_tmPWM8
LJ_tmPWM16
```

Sample file: ***LJGuideSamples\TimerPWM.ctl***

10.5.2 Period in

There are four Timer modes that allow you to measure the number of clock cycles between consecutive rising or falling edges. Two 16 bit, and two 32 bit. Using these modes is just a matter of performing all the basic steps we've described:

1) Enable a Timer:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 1, 0, 0)
```

2) Set the mode:

```
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmRISINGEDGE32, 0, 0)
```

The four modes are:

```
LJ_tmRISINGEDGES32
LJ_tmFALLINGEDGES32
LJ_tmRISINGEDGES16
LJ_tmFALLINGEDGES16
```

Note the plural form of Edge!

3) Set the clock frequency and divisor:

```
// use system clock so works on U3 and UE9:
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0)
```

4) GoOne() to actually execute the commands:

```
GoOne(ID)
```

5) Create a channel to read the Timer. **I/O Type** is **Timer**, **Channel #** is the timer #, in this case 0.

The difference between the rising and falling edge versions of these modes is self explanatory. The 32 bit versions allow you to measure longer lengths with higher clock frequencies, and thus higher resolution for long periods, but are subject to small errors because it is interrupt driven. If your lengths are short enough that the edges will always occur within 65535 clock cycles, you should use the 16 bit versions as they are not subject to the interrupt errors.

If you want to read the timer from script, you can use LJ_ioGET_TIMER:

```
private datain
eGet(ID, LJ_ioGET_TIMER, 0, @datain, 0)
```

Sample file: ***LJGuideSamples\TimerPeriodIn.ctl***

10.5.3 Duty cycle in

Duty cycle in is similar to setup as period in. The difference is that duty cycle in returns two values, the number of clock cycles the signal is high and the number of cycles the signal is low packed into one 32 bit number. These two values, therefore, are 16 bit, so you'll need to pick a clock frequency and divisor that won't overflow the 65535 counts possible. Setting up these modes is just a matter of performing all the basic steps we've described:

1) Enable a Timer:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 1, 0, 0)
```

2) Set the mode:

```
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmDUTYCYCLE, 0, 0)
```

3) Set the clock frequency and divisor:

```
// use system clock so works on U3 and UE9:
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0)
```

4) GoOne() to actually execute the commands:

```
GoOne(ID)
```

5) Create a channel to read the Timer. **I/O Type** is **Timer**, **Channel #** is the timer #, in this case 0.

The tricky part is actually parsing the data, since it is actually two different values packed into one number. The best way to do this is similar to the way we dealt with resetting counters, by creating extra, psuedo-channels to store the parsed data:

6) Create two more channels, one called **TimeHigh**, one called **TimeLow**. **Device Type** is **Test**, **D#** = **0**, **I/O Type** = **A to D**, **Chan #** = **0** and most importantly, **Timing** = **0**.

7) Click **Apply** to save your new channels, then click on the + next to **CHANNELS:** in the Workspace, then click on your Timer channel. We'll assume you called that channel RawDuty.

8) Click on the **Event** tab when the Channel view appears. Enter the follow script to parse the timer reading and click **Apply**:

```
TimeHigh.AddValue(RawDuty[0] % 0x10000)  // LSW
TimeLow.AddValue(floor(RawDuty[0] / 0x10000)) // MSW
```

This will split the single 32 bit reading into two separate readings and place them in their own channels.

Sample file: ***LJGuideSamples\TimerDuty.ctl***

10.5.4 Firmware counter in

This Timer mode works similar to a counter, but uses an interrupt routine to increment the counter so can't handle real high speed counts, and has a bit more internal overhead than a regular counter. Setting it up is basically the same as period in:

1) Enable a Timer:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 1, 0, 0)
```

2) Set the mode:

```
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmFIRMCOUNTER, 0, 0)
```

3) Set the clock frequency and divisor:

```
// use system clock so works on U3 and UE9:
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0)
```

4) GoOne() to actually execute the commands:

```
GoOne(ID)
```

5) Create a channel to read the Timer. **I/O Type** is **Timer**, **Channel #** is the timer #, in this case 0.

You can reset the timer to 0 by using LJ_ioPUT_TIMER_VALUE, put please read the section on resetting counters and how to get around it.

```
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 0, 0, 0, 0)
```

Sample file: ***LJGuideSamples\TimerFirmCount.ctl***

10.5.5 Firmware counter in w/ Debounce

This Timer mode works the same as Firmware Counter In, but introduces a debounce circuit for mechanical switch counting. It is really designed for frequencies less than 10hz, mostly push-button and reed-switch detection. Setting it up is similar to the regular Firmware Counter In, but has some extra steps to set the debounce settings:

1) Enable a Timer:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 1, 0, 0)
```

2) Set the mode:

```
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmFIRMCOUNTERDEBOUNCE, 0, 0)
```

3) Set the clock frequency and divisor:

```
// use system clock so works on U3 and UE9:
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0)
```

4) Set the debounce settings to a single 87ms period, positive edges counted:

```
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 0, 257, 0, 0)
```

5) GoOne() to actually execute the commands:

```
GoOne(ID)
```

6) Create a channel to read the Timer. **I/O Type** is **Timer**, **Channel #** is the timer #, in this case 0.

You can reset the timer to 0 by using LJ_ioPUT_TIMER_VALUE, put please read the section on resetting counters and how to get around it.

```
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 0, 0, 0, 0)
```

Sample file: ***LJGuideSamples\TimerFirmCount.ctl***

10.5.6 Frequency out

Frequency out is similar to PWM, but outputs a 50% duty cycle square wave. Because its fixed at 50% duty, a wider range of frequencies are attainable. Setup is similar to PWM, except the Timer value we specify is another divisor for the clock:

```
//Set the timer/counter pin offset to 0, which will put the first timer/counter on FIO4.
AddRequest (ID,  LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 4, 0, 0)

// use system clock so works on U3 and UE9:
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)

//Set the divisor to 24
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0)

//Enable 1 timer.  It will use FIO4.
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 1, 0, 0)

//Configure Timer0 as Frequency out.
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmFREQOUT, 0, 0)

//Set the second divisor to 10
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 0, 10, 0, 0)

//Execute the requests.
GoOne(0)
```

Sample file: ***LJGuideSamples\TimerFreqOut.ctl***

10.5.7 Quadrature

The quadrature Timer mode is designed explicitly for use with quadrature encoders. A quadrature encoder is a device that allows you to determine the absolute position of a rotating shaft. It does this by generating two pulses with each part of a rotation (how small of a rotation a "part" is depends on the encoder). One pulse will come before the other if the shaft is rotating in one direction, and the pulse order is flipped if the shaft is rotating in the other direction. The LabJack quadrature timer reads both these pulses and increments or decrements the timer reading depending on which pulse occurs first.

Because it takes two pulse signals coming in on two wires, the quadrature mode requires two timers, even though there is only one reading. The two timers have to be adjacent pairs, with the even timer as quadrature channel A, and the odd timer as quadrature channel B. Reading either timer returns the same, signed 32 bit count, and writing a zero to either timer resets both. Here's some DAQFactory script to initialize the quadrature mode on timers 0 and 1:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0)
//Configure Timer0 as quadrature.
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmQUAD, 0, 0)
//Configure Timer1 as quadrature.
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 1, LJ_tmQUAD, 0, 0)
GoOne(ID)
```

As you can see, its one of the easier timers to setup since it doesn't require the internal clock. The easiest way to

read the timer is to create a Timer channel: **I/O Type** is **Timer**, **Channel #** is the timer #, in this case 0, **Timing** can be whatever update interval you would like. Alternatively, you can use LJ_ioGET_TIMER to retrieve the reading from script:

```
private datain
eGet(ID, LJ_ioGET_TIMER, 0, @datain, 0)
```

10.5.8 Timer stop

Timer stop allows you to stop a particular (even numbered) timer after a certain number of pulses is received on the odd numbered timer stop timer pin. This is especially useful when used with frequency or PWM out to drive a stepper motor a certain number of pulses. For example, to generate exactly 1000 pulses on Timer 0, we'd setup timer 0 as frequency out, and timer 1 in timer stop mode and tie the two output pins together. You'll recognize the first part from the frequency out section:

```
//Set the timer/counter pin offset to 4, which will put the first timer/counter on FIO4.
AddRequest (ID,  LJ_ioPUT_CONFIG, LJ_chTIMER_COUNTER_PIN_OFFSET, 4, 0, 0)

// use system clock so works on U3 and UE9:
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_BASE, LJ_tcSYS, 0, 0)

//Set the divisor to 48.
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chTIMER_CLOCK_DIVISOR, 48, 0, 0)

//Enable 1 timer.  It will use FIO4.
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0)

//Configure Timer0 as Frequency out.
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmFREQOUT, 0, 0)

//Set the second divisor to 10
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 0, 10, 0, 0)

// now timer stop:
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 1, LJ_tmTIMERSTOP, 0, 0)

// set number of pulses:
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 1, 1000, 0, 0)

//Execute the requests.
GoOne(0)
```

Once the 1000 pulse are complete, Timer 0 will stop. To restart it, you'll need to reconfigure the timers by simply rerunning the above script.

Add a digital line to control the direction and you have a very easy stepper controller. But if you want it even easier, you can use a Channel event to allow a channel to trigger the pulses. To do this:

1) Create a sequence called PulseOut with the above script, replacing the 1000 in the last AddRequest with NumPulses[0]:

```
....
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 1, LJ_tmTIMERSTOP, 0, 0)

// set number of pulses:
AddRequest(ID, LJ_ioPUT_TIMER_VALUE, 1, NumPulses[0], 0, 0)
....
```

2) Create a new channel, call it **NumPulses**. **Device Type** = **Test**, **D#** = **0**, **I/O Type** = **D to A**, **Chan #** = a unique number (if you are using more than 1 Test D/A channel). Click **Apply**.

3) Click on the **+** next to **CHANNELS:** in the Workspace if not already expanded and click on the **NumPulses** channel.

4) Click on the **Event** tab, and put this script in:

```
beginseq(PulseOut)
```

Now, you can use the various DAQFactory components to simply set the NumPulses channel and the desired length pulse train will be outputted. Just remember that sliders and knobs will continuously update this channel and so are not good for changing NumPulses since the pulse train will likely take longer than the update speed. You can also change NumPulses in script:

```
NumPulses = 500
```

Just remember that as soon as NumPulses is set, the pulse train will start.

Sample file: ***LJGuideSamples\TimerStop.ctl***

10.5.9 System timer in

This mode allows you to read the free-running internal 64 bit system timer. The frequency of this timer is 750khz for the UE9 and 4MHz for the U3. Since DAQFactory's clock is precise to 1 microsecond (1 MHz), and there is a built in latency of a few milliseconds to actually read the LabJack, there are really only two uses for this timer. The first is when doing triggered stream. Here, DAQFactory has no way of determining the time of each scan, so we can use the system timer to apply a high precision time stamp as long as we include the timer in the stream. This is described in the section on Triggered Streaming.

The other use is when you need a high precision time stamp on another timer or counter read. Since all the timer and counter reads are done with a single call to the device, there is no software latency if the desired timers and the timer setup as system timer are read at the same time. This is as simple as making sure all your LJ_ioGET_TIMER_VALUE requests are together. This can also be achieved if you are using Channels to retrieve your timer readings as long as your timers all have the same Timing and Offset.

Depending on how long your experiment runs, you may be able to get away with only SYSTIMERLOW. For the UE9 at 750khz, the low timer will roll over every 5726 seconds, while the U3 at 4MHz rolls over in 1073 seconds. Here's the script to do both low and high system timers for longer experiments:

```
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chNUMBER_TIMERS_ENABLED, 2, 0, 0)
//Configure Timer0 as timer low.
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 0, LJ_tmSYSTIMERLOW, 0, 0)
//Configure Timer1 as timer high.
AddRequest(ID, LJ_ioPUT_TIMER_MODE, 1, LJ_tmSYSTIMERHIGH, 0, 0)
GoOne(ID)
```

Of course you'll probably have more than 2 timers enabled, or perhaps a counter or two.

If you actually need the high double-word of the system timer, you are going to end up with the time spread across two channels. You'll have to combine them. The easiest way is probably to use a calculated V channel. Here's how to do it, plus convert the counts to actual seconds:

1) Right click on **CHANNELS:** under **V:** in the Workspace and select **Add V Channel**. Give it a name, such as LJSystemClock.

2) In the Expression area enter the following and click Apply:

```
(SysTimerLow + SysTimerHigh << 32) / 4e6
```

This assumes you named your timer channels SysTimerLow and SysTimerHigh. It also assumes a U3 with a 4Mhz clock. For the UE9, change the 4e6 (4 million) to 750e3 (750 thousand).

You can now use this V channel anywhere you would a regular channel. You just have to prepend V. in front of the channel name:

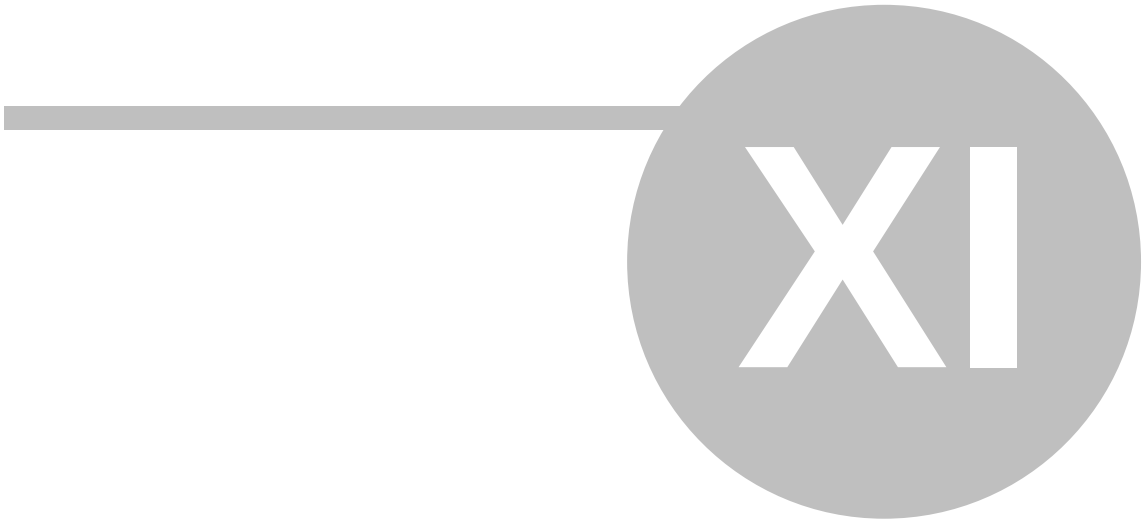
```
V.LJSystemClock
```

Sample file: see Triggered Streaming

Please note that DAQFactory uses 64 bit double precision floating point for all numbers. This representation has 52 bits of precision on the integer side, so you can only really store up to 52 bits of this counter. As the counter gets above 2 ^ 52, you will lose precision in the low order bits.

---

11 Advanced



11 Advanced

11.1 Opening a LabJack manually

There may be applications, most likely ones running in the Runtime version of DAQFactory, where you do not know the ID or address of the LabJack when creating the application and so can't hard code this in. Because of this, the LabJack driver has a function to allow you to manually open a LabJack, returning a device number that you can use with the rest of the functions just like before. For example, to open a UE9 at Ethernet address 192.168.2.1, you might do:

```
global DNum
private err
err = OpenLabJack(LJ_dtUE9, LJ_ctETHERNET, "192.168.2.1", 0, @DNum)
if (err != LJE_NOERROR)
   // failed to open
endif
```

This of course assumes you've done your using() and include() calls elsewhere already. In this example we used static parameters, but there is no reason you couldn't replace any or all of the parameters with variables that could be edited by the end user from pages you created.

Once successfully called, you should then use the DNum variable in all your other LabJack function calls.

A few points:

- There is no way to close a LabJack. This is handled automatically when you quit DAQFactory. That said, you don't want to go randomly opening LabJacks that don't exist as each attempt will use a little memory that can't be recovered until you quit DAQFactory.
- If you call OpenLabJack() with the exact same parameters, the function will not reopen the LabJack, but rather will return Device Number of the previously opened LabJack. Of course if the first call to OpenLabJack() failed, it will try again.

11.2 Raw In/Out and other functions that require array pointers

The Raw In and Raw Out functions (LJ_ioRAW_IN, LJ_ioRAW_OUT), and several other functions of the LabJack require an array pointer. You can pass a pointer to an array, just like you've been passing references to variables, using the @ sign. Just make sure you have preinitialized the array to the correct amount. So, using the example in the LabJack User's Guide for Raw In and Out, the DAQFactory script would look like this (assuming first found and that you've done the using() and include() somewhere else):

```
private writeArray = {0x70,0x70}
private readArray = {0x00,0x00}
private NumBytesToWrite = 2
private NumBytesToRead = 2
eGet(0, LJ_ioRAW_OUT, 0, @NumBytesToWrite, @writeArray)
eGet(0, LJ_ioRAW_IN, 0, @NumBytesToRead, @readArray)
```

Internally, DAQFactory will convert the array of double precision values, which is the only numeric data type supported in DAQFactory, to an array of bytes. This means that each element in the array should be between 0 and 255. The array also must be 1 dimensional. Most importantly, the array **MUST** be preinitialized to the proper length. The driver does not look at the previous parameter to make sure you have the correct array size, any more than the C version would do. If you do not preinitialize, you are likely to crash DAQFactory. Worse, it may work sometimes, but crash others, so be careful with this one.

11.3 SPI communications

The LabJack devices support doing serial communications using their digital lines using the standard SPI synchronous format. This is a powerful, but advanced feature of the LabJack, and the details of SPI is beyond the scope of this guide. Instead, here is some sample code that demonstrates SPI. It will use FIO0 for the clock (CLK), FIO1 for CS, FIO2 for MOSI, and FIO3 for MISO. It then sends out a string of 16 bytes and prints the received string to the command / alert window. If you short MISO to MOSI, then the 16 bytes sent out are echoed back. If MISO is tied to GND, then all zeros are received and printed. If you tie MISO to VS or leave it unconnected, then all 255's are received and printed. Here's the script. Its pretty much a direct copy of the C sample provided by LabJack:

```
using("device.labjack")
include("c:\program files\labjack\drivers\labjackud.h")
global ID = 0 // use first found

//First, we do a pin config reset to set the LabJack to factory defaults.
ePut(ID,LJ_ioPIN_CONFIGURATION_RESET,0,0,0)

// Configure the SPI communication:
//Enable automatic chip-select control.
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSPI_AUTO_CS,1,0,0)
//Mode A:  CPHA=1, CPOL=1.
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSPI_MODE,0,0,0)

//125kHz clock.
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSPI_CLOCK_FACTOR,0,0,0)

//MOSI is FIO2
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSPI_MOSI_PIN_NUM,2,0,0)

//MISO is FIO3
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSPI_MISO_PIN_NUM,3,0,0)

//CLK is FIO0
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSPI_CLK_PIN_NUM,0,0,0)

//CS is FIO1
AddRequest(ID, LJ_ioPUT_CONFIG, LJ_chSPI_CS_PIN_NUM,1,0,0)

//Execute the requests on a single LabJack.  The driver will use a single low-level TimerCounter command to hand
GoOne(ID)

// now that its setup, do the communication.  Note that you can do this part in a separate sequence, and run mul
// without the reconfiguring the SPI with the above code.

// initialize the variables
private numSPIBytesToTransfer=4
private dataArray
dataArray[0] = 170
dataArray[1] = 138
dataArray[2] = 85
dataArray[3] = 21

//Transfer the data.  The write and read is done at the same time.
eGet(ID, LJ_ioSPI_COMMUNICATION, 0, @numSPIBytesToTransfer, @dataArray)

// print the read to the command / alert window.  Of course you'll probably do something a bit more exciting wit
? dataArray
```

Each time you run the script the 4 bytes of dataArray will be written and then 4 bytes will be read back and printed to the command / alert window. As mentioned in the script comments, you can do the actual communication multiple times without re-running reconfiguration script at the top of this sample.

11.4 Utilizing multicore processors

In many places in this guide, we've split apart things that occur at less than 100hz, and those that are faster. This is because Windows needs to have some CPU time to redraw the screen, move the mouse and perform other tasks and thus can't really do things at an interval faster than 100hz.

However, if you have a multicore or multiprocessor computer, you can take advantage of DAQFactory's design and the multiple cores to achieve faster software polled rates than would be possible with a single core. The trick to this is making sure that all your fast processes, i.e. anything faster than 50hz or so, is done in a single thread.

What's a thread? You've seen Windows do multiprocessing before when you are downloading a big file off the internet and you switch to another program to check your email at the same time. A thread is just like different programs that can run simultaneously, but that exist inside of a single program like DAQFactory. DAQFactory is made up of a lot of different threads and so can perform multiple tasks simultaneously. Likewise, it can split these threads across multiple cores or processors for maximum efficiency.

There are basically two ways you can keep things on a single thread in DAQFactory. The first is to put all the tasks in a single sequence. Each sequence is run in its own thread, unless of course it is called as a function from another sequence. This is why you can create multiple sequences to perform different tasks and run them all simultaneously. The other way is to create Channels, and give all the desired high speed channels the same Timing and Offset values. If Channels have different Timing or Offset, then they are put on a different thread.

If you have multiple processors or cores and want to see this in action, just create a few analog input channels and set their Timing to 0.001. This will chew up much of the processor power of one of your cores, but will leave Windows the other core to perform its tasks. In fact, as I am typing this, I have DAQFactory running reading an analog input from a LabJack at full speed, and I see no lag in my typing.

That all said, its important to understand that there is a limit to how fast the LabJack itself can process commands. If you perform the test with a few Channels with Timing = 0.001, and then go to the Table tab of one of the channels, you will see that the data actually comes in about every 4 milliseconds. This is because although DAQFactory is trying to read it at 1 millisecond intervals, the LabJack is taking about 4 milliseconds to actually perform the command.

11.5 Unsupported functions

The LJ_ioSET_STREAM_CALLBACK, LJ_ioSET_EVENT_CALLBACK and any other LabJack function that requires a function pointer are not supported. These particular two callback's are handled internally by DAQFactory. The first allows DAQFactory to process streaming data, while the second handles connect and disconnect messages from the driver.

Also, you should not change the LJ_chSTREAM_WAIT_MODE, as all waiting is handled internally. If you change this, you will most likely cause streaming to stop functioning.